

# **BASIC 4.0**

# **Programming**

# **for the**

# **Commodore**

# **PET<sup>®</sup> / CBM<sup>™</sup>**

- A comprehensive step-by-step guide to basic programming including graphics, animation, sequential files and debugging procedures.
  - Includes thirty-seven programs to help you get the most out of your computer.
- For both cassette and disk drives.



**DON CASSEL**

***BASIC 4.0  
Programming  
for the  
Commodore  
PET<sup>®</sup>/CBM<sup>™</sup>***



MICROPOWER SERIES

***BASIC 4.0  
Programming  
for the  
Commodore  
PET<sup>®</sup>/CBM<sup>™</sup>***

***DON CASSEL***  
***Humber College***

**wcb**

Wm. C. Brown Publishers  
Dubuque, Iowa

Cover photo is provided courtesy of Commodore Business Machines.

Figures 1.1, 2.1, 2.2, 2.5, 2.6, 8.1, 8.2, and Appendix F are provided courtesy of Commodore Business Machines.

Edouard J. Desautels, University of Wisconsin-Madison  
Consulting Editor

Copyright © 1983 by Wm. C. Brown Company Publishers. All rights reserved

Library of Congress Catalog Card Number: 83-70351

ISBN 0-697-08265-2

2-08265-01

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Printed in the United States of America.

***To Pamela***



# **Contents**

---

*Preface xiii*

## **1**

**Introduction to the PET/CBM 1**

**HARDWARE CONFIGURATION 1**  
**MEMORY 2**  
**BASIC 4.0 3**  
**DOS 2.0 4**  
**REVIEW QUESTIONS 4**

## **2**

**Getting Started on the PET/CBM 5**

**PET MODELS 5**  
**TURNING ON THE PET 5**  
**KEYBOARD CHARACTERS 7**  
    **PET Graphics Keyboard 7**  
    **CBM Business Keyboard 8**  
**USING THE CASSETTE TAPE 9**  
    **Inserting a Cassette 9**  
    **Tape Controls 10**  
    **Loading a Program 10**  
    **Saving a Program 11**  
**USING THE FLOPPY DISK DRIVE 11**  
    **Loading a Program 12**  
    **Saving a Program 12**  
**DIRECTORIES 12**  
**BACKUPS 13**  
**REVIEW QUESTIONS 13**

## **3**

**Elementary BASIC Programming 15**

**CALCULATOR MODE 15**  
**HOW TO USE CURSOR CONTROLS 16**  
**INSERT/DELETE 16**  
**CLEAR/HOME 17**



<b>RUN/STOP</b>	17
<b>BASIC NUMBERS</b>	18
Integers	18
Real Numbers	18
Scientific Notation	20
<b>STRINGS</b>	21
<b>VARIABLES</b>	21
<b>MULTIPLE STATEMENTS PER LINE</b>	22
<b>ARITHMETIC STATEMENTS</b>	23
Add (+)	23
Subtract (—)	23
Multiply (*)	23
Divide (/)	24
Exponentiation (↑)	24
<b>HIERARCHY AND PARENTHESES</b>	24
<b>PRINT</b>	25
<b>INPUT</b>	27
<b>GOTO</b>	28
<b>REM</b>	29
<b>SIMPLE CALCULATION PROGRAM</b>	29
<b>FAHRENHEIT—CELSIUS PROGRAM</b>	30
<b>IMPROVING YOUR SOLUTION</b>	31
<b>WEIGHTED AVERAGE PROGRAM</b>	32
<b>COMPUTING LOAN PAYMENTS</b>	35
<b>REVIEW QUESTIONS</b>	36

## **4**    **Not So Basic BASIC** 39

<b>DIM</b>	39
Storing Values in an Array	40
Multi-Dimensional Arrays	41
<b>IF—THEN</b>	42
<b>FOR—NEXT</b>	44
<b>GOSUB—RETURN</b>	46
<b>STOP</b>	47
<b>GENERATING RANDOM NUMBERS</b>	48
<b>NUMBER GUESSING GAME</b>	49
<b>TIME DELAYS</b>	50
<b>IMPROVED WEIGHTED AVERAGE PROGRAM</b>	52
<b>CALCULATING TRIP COSTS</b>	53
<b>REVIEW QUESTIONS</b>	56

## **5**    **More on Input and Output** 59

<b>READ—DATA</b>	59
<b>CREATING A BAR CHART</b>	60
<b>WEIGHTED AVERAGE WITH DATA</b>	61
<b>RESTORE</b>	62
<b>MORE ABOUT PRINT</b>	63
Cursor Controls	63
TAB	65
SPC	65
<b>METRIC CONVERSION PROGRAM</b>	66
<b>SIMPLE PAYROLL PROGRAM</b>	70
<b>REVIEW QUESTIONS</b>	75

# 6

## **Advanced BASIC 77**

<b>GET</b>	77
<b>ON—GOSUB</b>	78
<b>ON—GOTO</b>	79
<b>POKE</b>	80
<b>PEEK</b>	82
<b>TI AND TI\$ FUNCTIONS</b>	83
<i>TI or Time Function</i>	83
<i>Reaction Timer</i>	84
<i>TI\$ or Time\$</i>	84
<b>DEF FN</b>	85
<b>ARITHMETIC FUNCTIONS</b>	87
<b>ABS</b>	87
<b>ATN</b>	87
<b>COS</b>	87
<b>EXP</b>	88
<b>INT</b>	88
<b>LOG</b>	88
<b>RND</b>	88
<i>Creating a Specific Set of Random Numbers</i>	89
<b>SGN</b>	89
<b>SIN</b>	89
<b>SQR</b>	90
<b>TAN</b>	90
<i>Converting Radians to Degrees</i>	90
<b>STRING FUNCTIONS</b>	90
<b>ASC</b>	90
<b>CHR\$</b>	91
<b>LEFT\$</b>	91
<b>LEN</b>	92
<b>MID\$</b>	92
<b>RIGHT\$</b>	94
<b>STR\$</b>	94
<b>VAL</b>	94
<b>CONCATENATION</b>	94
<b>PLOTTING GRAPHS</b>	95
<b>CONTROLLING DECIMAL POSITIONS</b>	98
<i>Driver Routines</i>	98
<b>CAI CHESS AND PROGRAM GENERALIZATION</b>	99
<i>The Data</i>	101
<i>The Program</i>	102
<b>REVIEW QUESTIONS</b>	108

# 7

## **Interacting with the User of Your Program 109**

<b>USER LEVEL</b>	109
<i>Casual</i>	109
<i>Trained</i>	109
<i>Programming Skills</i>	109
<b>USER DIALOGUES</b>	110
<i>Prompting</i>	110

**DEFAULT RESPONSES 111**  
**MENUS 112**  
**MULTILEVEL MENUS 113**  
**FORM FILLING 114**  
**COMMAND LANGUAGES 115**  
**REVIEW QUESTIONS 116**

## **8**

**Graphics, Animation, and Sound 117**

**GRAPHIC CHARACTER SET 117**  
**USING PRINT TO PRODUCE A GRAPHIC 120**  
    *Graphic of the PET Keyboard 120*  
    *Reaction Timer With Graphics 120*  
**USING POKE TO PRODUCE A GRAPHIC 121**  
    *Lunar Lander 122*  
    *Chessboard 123*  
**ANIMATION 123**  
    *Rocket 1 123*  
    *Rocket 2 124*  
    *Egg Timer 125*  
    *Rolling Die 127*  
**GENERATING SOUNDS 129**  
**MUSIC PLAYER 131**  
**REVIEW QUESTIONS 132**

## **9**

**Tape Files Extend Your Reach 133**

**CONCEPTS 133**  
**OPEN, CLOSE 134**  
**PRINT# 135**  
**WRITING BUDGET NAMES ON TAPE 135**  
**INPUT# 136**  
**READING THE BUDGET NAMES TAPE 137**  
**HOW TO HANDLE RECORDS WITH MULTIPLE FIELDS 138**  
**UPDATING A TAPE FILE 139**  
**SUMMARY OF TAPE FILES 139**  
**REVIEW QUESTIONS 142**

## **10**

**Disk Files 143**

**SEQUENTIAL FILES 144**  
    *DOPEN and DCLOSE 144*  
    *PRINT# and INPUT# 145*  
    *Writing Budget Names on Disk 145*  
    *Reading Budget Names from Disk 146*  
    *How to Handle Multiple Fields on Disk Files 147*  
    *Detecting Disk Errors 147*  
**MAINTAINING THE CHECKBOOK 149**  
    *Replacing a Current File 149*  
    *Using Variable File Names 149*  
    *English Code 150*  
    *Display Screen Page 151*  
    *Deleting Table Entries 151*

<b>RELATIVE ACCESS FILES</b>	<b>155</b>
<i>Creating a Relative File</i>	157
<i>Accessing a Relative File</i>	157
<i>Updating a Relative File</i>	158
<i>Appending Records to an Existing File</i>	159
<i>How to Use an Index File</i>	160
<b>REVIEW QUESTIONS</b>	<b>162</b>

# 11

## ***How to Debug Your Programs*** 165

<b>DESK CHECKING</b>	<b>166</b>
<b>SYNTAX ERRORS</b>	<b>166</b>
<b>TEST DATA PREPARATION</b>	<b>167</b>
<b>IMMEDIATE MODE DEBUGGING</b>	<b>168</b>
<b>TRACING PROGRAM LOGIC</b>	<b>169</b>
<i>Manual Tracing</i>	169
<i>Automatic Program Tracing</i>	172
<b>REVIEW QUESTIONS</b>	<b>178</b>

## ***Appendices*** 179

<b>A BASIC OPERATING COMMANDS</b>	<b>179</b>
<b>APPEND</b>	179
<b>BACKUP</b>	179
<b>CATALOG</b>	180
<b>CLR</b>	180
<b>COLLECT</b>	180
<b>CONCAT</b>	180
<b>COPY</b>	181
<b>DIRECTORY</b>	181
<b>LIST</b>	181
<b>LOAD</b>	181
<b>NEW</b>	182
<b>RENAME</b>	182
<b>RUN</b>	182
<b>SAVE</b>	183
<b>SCRATCH</b>	183
<b>VERIFY</b>	183
<b>B RESERVED WORDS</b>	<b>185</b>
<b>C ABBREVIATIONS</b>	<b>187</b>
<b>D DOS ERROR MESSAGES</b>	<b>189</b>
<b>E PET ASCII AND PEEK/POKE</b>	
<b>CODES</b>	191
<b>F ASCII AND CHR\$ CODES</b>	<b>193</b>

<b>Credits</b>	<b>197</b>
<b>Index</b>	<b>199</b>



## ***Preface***

---

**T**his book is an introduction to the BASIC language used on the Commodore PET and CBM micro-computers. If you are a beginner at programming in the BASIC language, this material will provide you with a gradual introduction to BASIC stressing a hands-on approach with the PET or CBM.

The BASIC used in this book is BASIC 4.0 released by Commodore since 1981. If your computer does not have BASIC 4.0, don't worry because the major part of the language is unchanged from previous releases of BASIC. Most of the programs in the book will still work on earlier PETs.

A floppy disk is available with this book. It will provide you with the programs presented throughout each chapter and will be especially valuable to you if the program under discussion is first loaded from the disk. In cases where small segments of program code are discussed, it would be helpful to type the code into your computer and RUN it so that you gain a complete understanding of the example.

The first chapter is a very brief introduction to the PET/CBM hardware with BASIC 4.0 and DOS 2.0 for the floppy disk. Chapter 2 covers the use of the keyboards, tape, and floppy disk. If you have used the PET or CBM before, you may skip over these chapters or read through them quickly.

Chapter 3 begins with an elementary introduction to the BASIC language, stressing a hands-on approach. By entering the program code on the computer as you read, you will get immediate feedback from your computer. Enough BASIC statements are introduced in this chapter so that you can begin to write some useful programs at this early stage.

Chapter 3 ends with five complete programs that may be run on either the PET or the CBM. In these five programs, two things are emphasized. One is the need for planning a program. The approach taken shows how the input and output need to be defined before the program is written and then shows how to make use of an English code (Pseudo code) to develop the general program logic. Later the concept of flowcharts is discussed.

The second emphasis in the sample programs is how to apply the language statements just discussed in the chapter to a variety of situations. Each program is designed to expose you to realistic situations that require the use of the BASIC statements you have just studied.

Subsequent chapters follow the same pattern as chapter 3, but go into more depth in the language, as indicated in the Contents. Each chapter has numerous examples and ends with several programs (included on the disk) that apply the new features of the language. Each program has been completely developed using the techniques of program design to develop the program logic.

Chapter 7 deviates slightly from this pattern and covers ways that we can communicate with users of our programs. Since you will sooner or later write programs for other people to use, this chapter considers effective interaction with users of your program and how to implement these concepts on the PET or CBM.

Next we will look at graphics, animation, and sound in chapter 8. These topics will be particularly interesting to PET users, although they also apply to the CBM. Chapter 9 covers the use of sequential files on tape and chapter 10 explains both sequential and relative files on disk. Again, there are examples and programs to try so that the concepts are made clear.

Finally, chapter 11 contains the procedure for debugging your programs. As you will discover, programs usually don't work immediately after you have written them. In fact, having a perfectly working program the first time is the exception rather than the rule. So in this chapter, some techniques for finding your bugs and correcting them are discussed.

As I mentioned earlier, this is an introduction to BASIC programming on the PET/CBM. This book does not pretend to be an exhaustive treatment of programming and there is much more to learn about the PET/CBM beyond this level. However, I hope you will find this book instructive and helpful as you learn to program and I trust it will provide the necessary foundation for you to move on to more advanced programming on the PET/CBM.

I want to thank the reviewers of this book, Edouard J. Desautels, University of Wisconsin at Madison and Alfred Shin, Humber College at Ontario, Canada.

Don Cassel

---

---

# 1

---

---

## ***Introduction to the PET/CBM***

---

**T**his book is about programming the PET/CBM microcomputer using the BASIC 4.0 language. Since most of the BASIC language is unchanged from BASIC 1.0 to 4.0 the majority of the discussion in this book will apply to your PET even if you don't have level 4.0 BASIC.

Possibly you have purchased your PET/CBM to solve a specific problem in your business, school, or home. Many people have used the PET/CBM successfully for solutions to problems ranging from accounting to computer-assisted instruction, financial management to developing music skills, personal income tax calculation to simulations. The list is endless. However, before you can apply the computer to your problems it is necessary to learn to use the tool effectively. After studying the contents of this book you will be equipped to solve many of your programming applications with the BASIC language. But first let's look at the hardware.

### ***HARDWARE CONFIGURATION***

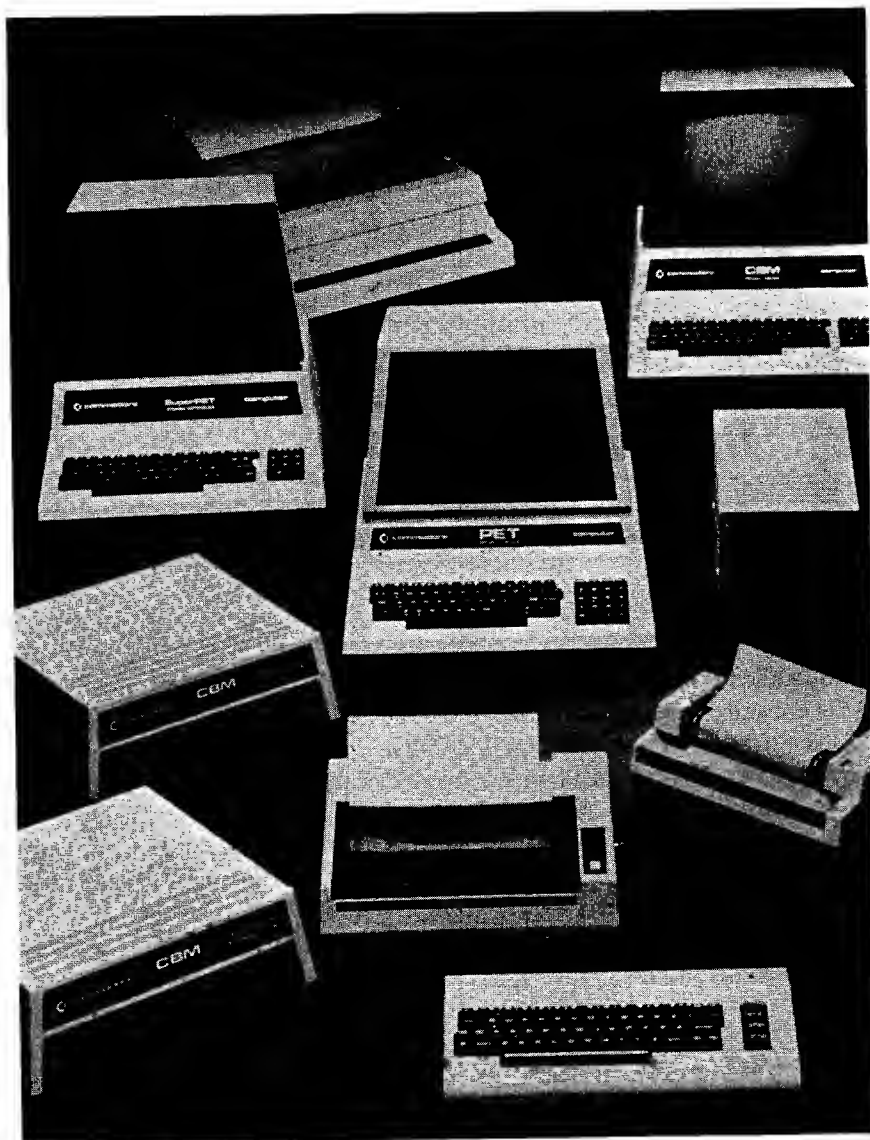
Figure 1.1 shows PET, CBM, and SuperPET microcomputers each with a display screen and a keyboard. Also in this figure are floppy disk drives and printers used with the Commodore computers. Hidden inside the computer's case are the electronic components—a microprocessor and memory—necessary for its operation.

To understand the basics of the PET/CBM it is useful to compare the computer to parts of the human body. Of course, the similarity is only coincidental. For example, when you want to learn something new this is often done by reading a book such as you are doing right now. The new information passes through your eyes and becomes input to your brain, where it is stored in memory. Similarly new information can be entered into the computer through the keyboard and then stored temporarily in the PET's electronic memory or storage.

After you have read something you may think about it by processing it mentally in your conscious brain. Then you might discuss your conclusions with someone or maybe simply repeat verbally what you have read. Your voice is then output from the brain. When information has been processed by the computer's processor (brain?) then that same information or a new arrangement of it may then be displayed on the screen, which is the computer's output device.

These physical components of the computer: display, keyboard, processor, and so forth are called the hardware. The program that is used to describe the processing steps is called the software. The program resides in the computer's memory when it is being used and describes in very precise steps how the computer is to process the data.



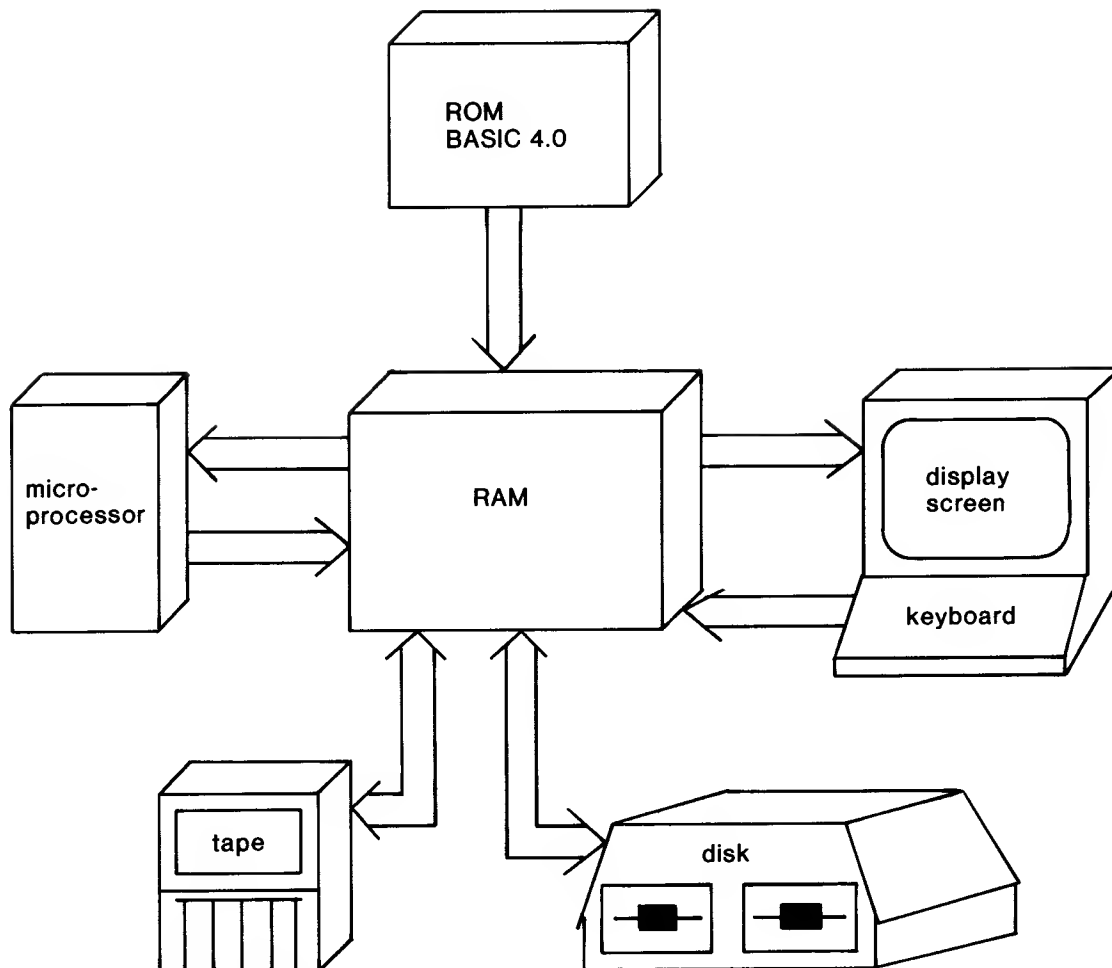


**Figure 1.1** PET and CBM hardware configuration

If your brain is anything like mine you will not likely remember everything you have read. One solution to this is to make notes by writing or typing. This process is another kind of output. Although the computer never forgets anything entered into its memory it does have limited capacity. Therefore another form of output is often used to record or store information outside of the computer. One form of output is the tape cassette. Another is the floppy disk or diskette. Information recorded on the tape or disk can be stored and read back into the computer at a later time when it is needed, just as you can read your notes to jog your memory. Tape and disk are commonly used to store programs as well as other data.

## **MEMORY**

When you type something on the keyboard or read from tape or disk the information goes into the PET's memory or RAM (Random Access Memory) as shown in figure 1.2. RAM is a solid state memory device that is a part of the PET's circuitry. Programs or data may be read or written into RAM. Normally only one program at a time goes into memory and when a new program is read from tape or disk it will replace the previous program in memory.



**Figure 1.2** Components of a PET/CBM system

Common memory sizes in the PET/CBM are 8K, 16K, or 32K, where K represents 1024 characters or, in computer lingo, bytes. The memory size of your computer will determine the size of program it is capable of running. If you have 8K of memory the computer will not have room for a 16K or 32K program.

Also shown in figure 1.2 is a component called ROM for Read Only Memory. As the name suggests, ROM may only be *read* by the computer. The most significant part of ROM is the BASIC interpreter it contains. This interpreter is a program that has been prerecorded in the PET's ROM and is used by the computer to run your BASIC programs.

### **BASIC 4.0**

BASIC is the primary language for programming the PET/CBM. It is through BASIC that you give instructions to the computer to solve a particular problem. While all levels of BASIC on the PET/CBM are essentially the same, level 4.0 has corrected some errors that had existed on previous machines. In addition, BASIC 4.0 includes some new commands that can make your life easier as a programmer. These commands will be discussed at the appropriate time.

## **DOS 2.0**

DOS (*for Disk Operating System*) is the program in the floppy disk drive that controls the reading and writing of disk files. DOS 2.0 is the software released by Commodore to work with BASIC 4.0. Any discussion of disk files will assume the use of DOS 2.0 on your computer.

### **REVIEW QUESTIONS—CHAPTER 1**

1. Name some of the components of a PET or CBM microcomputer.
2. Compare the concepts of input, process, and output to the human body. Can you think of any other analogy to compare with the computer?
3. What is hardware? Give some examples.
4. What is software? What purpose does it perform in the microcomputer?
5. What is meant by the term RAM? What sizes of RAM are common for the PET and CBM microcomputers?
6. Discuss the function of BASIC on the PET/CBM. What level of BASIC does your microcomputer have?
7. What device must your computer have if you will need to use DOS 2.0?

# 2

## **Getting Started on the PET/CBM**

**I**f you have already used a PET then you might wish to skip ahead to chapter 3. Otherwise you should read this chapter to learn the basic operation of the PET/CBM.

### **PET MODELS**

The PET/CBM comes in two basic models: the PET (Personal Electronic Transactor) and the CBM (Commodore Business Machine). The PET (figure 2.1) comes with the fullsize graphics keyboard, a 40-column screen, and is available with 8K, 16K, or 32K RAM. The most basic configuration for this model is with a cassette tape, while the floppy disk and/or printer are popular additions.

The CBM with the 80-column screen (figure 2.2) comes with 32K to 96K of RAM. It is designed specifically for the business user and does not have graphics on the keyboard. Rather the keyboard selects either upper or lower case letters, numbers, or special characters. The CBM is generally used for word processing, accounts payable and receivable, and inventory applications.

### **TURNING ON THE PET**

First be sure the PET's line cord is plugged into a 3-prong wall outlet. If only a 2-prong outlet is available an adapter will need to be used to convert the 3-prong electrical cord on the PET to a properly grounded 2-wire system.

Now reach behind the left rear panel of the PET and press the power switch to the ON position. In a few seconds the screen will come to life and display the following characters:

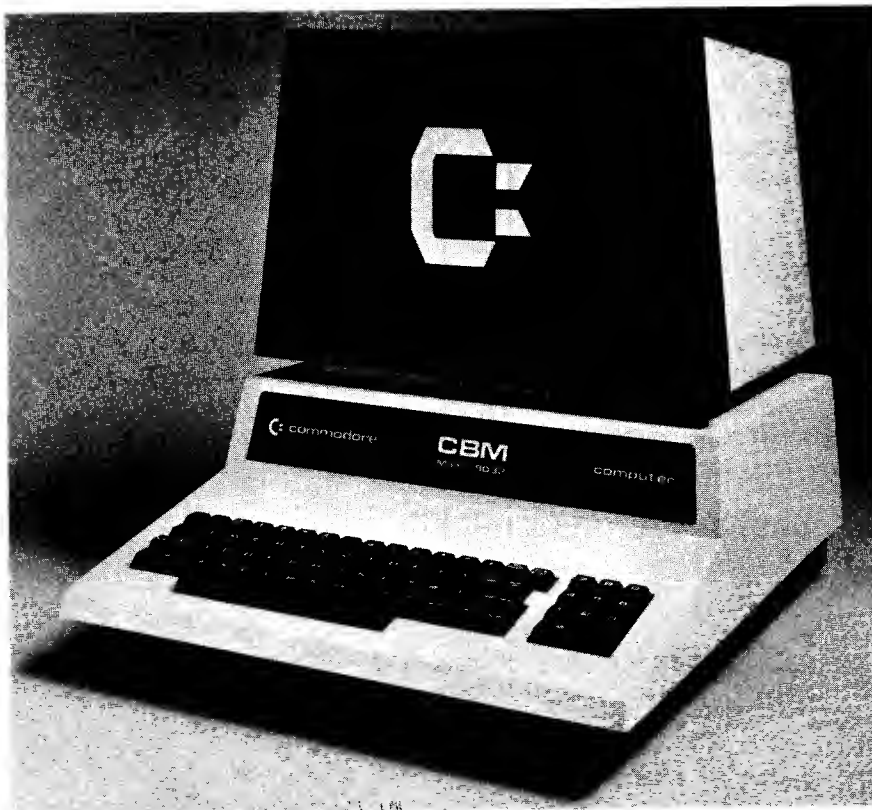
```
*** COMMODORE BASIC 4.0 ***
31743 BYTES FREE
READY.
```

Don't be alarmed if your PET's display is slightly different. These differences can be due to the size of memory or, on older machines, the level of the ROMs currently in the computer. Revision 2 ROMs will display:

```
*** COMMODORE BASIC ***
```



**Figure 2.1** Commodore PET model 4032



**Figure 2.2** Commodore CBM model 8032

revision 3 ROMs show:

```
### COMMODORE BASIC ###
```

and revision 4 ROMs show:

```
*** COMMODORE BASIC 4.0 ***
```

ROM size differences will cause the BYTES FREE message to appear something like the following:

```
4K    3071 BYTES FREE
8K    7167 BYTES FREE
16K   15359 BYTES FREE
32K   31743 BYTES FREE
```

## KEYBOARD CHARACTERS

### PET Graphics Keyboard

The keyboard for the PET is illustrated in figure 2.3. This keyboard is arranged with 74 keys for alphabetic, numeric, special, and graphic characters. The main keyboard contains 54 keys for upper-case alphabetic letters from A to Z and special characters such as \$ % " , ( ) that are mainly used as special programming symbols. This keyboard also contains most of the graphic symbols on the front of each key. To select the graphic symbol it is necessary to hold down one of the two shift keys or the shift/lock while you also press the appropriate graphic key.

At the right of the main keyboard is a Return key. Whenever an entry is made to the PET, such as a BASIC statement or a response to a question from a program, the Return key is pressed to enter that line into memory.



Figure 2.3 PET graphics keyboard

Beside the Return key is a RUN/STOP key. This key may be used to STOP a program or to RUN (by holding shift and pressing RUN) a program from disk.

On the left of the main keyboard is an OFF/RVS (reverse) key. When this key is first pressed, reverse is on. Try this and type some characters. Notice that the character is now dark on a bright background. Now hold shift and press RVS. This turns reverse OFF. Now type the same characters and notice the difference.

The key pad to the right contains numbers and some more graphic characters. Along the top of this number pad are four cursor control keys used to control cursor movement.

### ***CBM Business Keyboard***

The fullsize keyboard for the CBM (figure 2.4) is designed primarily for business use and as a result does not have the graphic characters. On the CBM graphics may be initiated by typing

**POKE 59468,12**

Now all upper case will become graphics characters. To return to upper/lower case alphabetic, type

**POKE 59468,14**

Both the PET and CBM keyboards have four cursor control keys. Try pressing the down-pointing arrow and then the right-pointing arrow to see what happens. Notice how the cursor moves in the direction of the arrow each time you press the key. Now press the HOME key. The cursor should return immediately to the upper left corner or home position of the screen.



**Figure 2.4** CBM business keyboard

## **USING THE CASSETTE TAPE**

If you will be using tape to store and load your programs you will want to read this section. However, if you have the disk drive skip this section and go on to the section using disk which follows.

The cassette drive (figure 2.5) is a convenient low-cost way to load a program that has been previously written on the PET and saved on cassette tape. The PET's RAM can only hold one program at a time, but the only limit to the number of programs you may have is dependent upon the number of cassettes and their length.

When purchasing tapes for use on the PET it is best to use relatively short tapes, C-10 to C-30, and store only one or two programs per side. If you store many more programs on a single tape the waiting time for a program to load becomes prohibitive. Tapes should be of the low noise, high quality, and high output variety. Avoid the cheap brands.

### **Inserting a Cassette**

Inserting a tape in the PET's drive is no different from using a music or audio cassette. First press the STOP/EJECT key to open the door. Now insert the tape with the open end toward you and the label facing up. If it won't go in, don't force it. You are probably putting it in the wrong way; so turn it around and try again. Now press the door closed.



**Figure 2.5** Cassette tape drive



## ***Tape Controls***

The tape controls are basically like other cassettes except the PET's tape does not have volume or tone controls. The controls you will need are:

1. REC—This is the record key, which is used simultaneously with the play key when you want to save a program from the PET's RAM onto a cassette tape.
2. REW—The rewind key is used to ensure that the tape begins at the leader before any program is reached. It will also be used to rewind the tape after the desired program has been read.
3. FFWD—The fast forward key lets you advance the tape.
4. PLAY—This key will be pressed when a program is to be loaded into RAM from the tape.
5. STOP/EJECT—This key can be used to stop reading or saving but is normally used to eject the tape when you are finished with it.

## ***Loading a Program***

To load a program from tape simply type the command

**LOAD**

and press Return. If the name of the program is known, such as ACCOUNTS, then you could type

**LOAD"ACCOUNTS"**

and press Return.

In the first case the computer would simply look for the first program on the tape and load it. When you supply the program's name the computer will only load a program with that identical name and will bypass any other program it happens to find on the tape.

After you pressed Return the PET will respond with

**PRESS PLAY ON TAPE #1**

When you press the play button it then displays

**OK  
SEARCHING**

after a few seconds it should display

**FOUND ACCOUNTS  
LOADING**

or a different program name depending on the tape you are using. Once the program is loaded, which might take as long as a minute, the message

**READY**

will appear. If you get a Load Error, rewind the tape and start again from where we typed LOAD.

Now type

**RUN**

and press Return. Your program will now begin to run.

### ***Saving a Program***

When you start writing your own programs they may be saved for future use on a cassette tape. First the program must be in the PET's memory. This could be a program you have typed in or it could have been loaded from another tape. The PET really can't tell the difference. To save the program, type

**SAVE"ACCOUNTS"**

This means we are naming the program ACCOUNTS. Make sure a blank cassette is in the drive and then press Return on the PET. The PET displays

**PRESS PLAY AND RECORD ON TAPE #1**

Do this, being sure both buttons are pressed simultaneously. The PET says

**OK  
WRITING ACCOUNTS**

When it is finished saving the program the message

**READY.**

will appear. To ensure the save worked you can now rewind the tape and type VERIFY. Press Return and follow instructions. The PET now checks each character on the tape with each character in its memory. If all went well the message

**OK  
READY.**

will appear on the screen.

### ***USING THE FLOPPY DISK DRIVE***

The advantage of having a disk drive (figure 2.6) is its greater speed in reading programs or data. A program that takes a minute to load from tape will require only two or three seconds from disk.



**Figure 2.6** Dual drive floppy disk

## ***Loading a Program***

To use disk insert a diskette (with the oblong or oval-shaped opening pointing away from you) containing the program you want to run into drive 0 and gently close the door. If the program you want is the first one on the disk then simply hold down Shift and press RUN/STOP. This action will both load and run the program.

If the program is not first on the disk use the following command format:

**DLOAD"program name"**

If your program is called BUDGET then type

**DLOAD"BUDGET"**

If your program happened to be on a disk in drive 1 (the left drive) then the format to use is

**DLOAD"program name",D1**

## ***Saving a Program***

To save a program on diskette from the PET/CBM's memory use the DSAVE command.

**DSAVE"program name"**

for drive 0 or

**DSAVE"program name",D1**

for drive 1.

After you have saved programs on the floppy disk place an adhesive tape over the notch on the floppy's container. This tape will protect the disk from writing over your programs or data accidentally. If you wish to add additional programs at a later time the tape must first be removed before inserting the diskette into the drive.

To replace a program precede the program name with the at sign as follows:

**DSAVE "@ACCOUNTS"**

## ***DIRECTORIES***

Diskettes also contain directories, which list their contents. If at any time you want to know what is on a disk, list the directory of drive 0 with the command:

**DIRECTORY D0**

and drive 1 with

**DIRECTORY D1**

The name CATALOG may be used interchangeably with DIRECTORY.

During the display of a directory the scrolling may be stopped by pressing the Space bar (: on the CBM) to let you more easily read its contents. Pressing space again (9 on the CBM) will continue the display.

## **BACKUPS**

When you have stored a number of programs on disk it is possible occasionally to destroy them. Usually this happens inadvertently, but the loss can be rather shattering especially if it represents hours of your time. Keeping a second copy on another diskette as a backup is well worth the small amount of time and expense necessary. The COPY and BACKUP commands are used for this purpose. These commands are discussed in Appendix A.

## **REVIEW QUESTIONS—CHAPTER 2**

1. Describe the basic differences between the PET and CBM computers.
2. When the power to the PET is turned on what is meant by the message "31743 BYTES FREE"?
3. How are the graphic characters on the front of the keys selected on the fullsize graphics keyboard?
4. What is the purpose of the RUN/STOP key?
5. How does the OFF/RVS key work?
6. Explain the commands for loading a program from tape into memory and running it.
7. If you have a program currently in memory how could this program be saved on a cassette tape with the name PET BOOKS?
8. Explain how to load and save programs to disk using the name PET BOOKS.
9. What is the purpose of a directory? How can you get one from drive 1?



# 3

## **Elementary BASIC Programming**

**W**hat is BASIC? The name BASIC is a copyrighted term of the trustees of Dartmouth College and is an acronym for Beginner's All-purpose Symbolic Instruction Code. BASIC was developed by Professors John G. Kemeny and Thomas E. Kurtz on a time-sharing computer system in 1963. BASIC was designed with the purpose of making programming an easier task for the average person who had little computer background or knowledge.

As the name implies, BASIC is relatively easy to learn but since 1963 many new enhancements have been added to the language, extending its usefulness. These developments have led to many new versions of the language, of which Microsoft BASIC for the PET is one. Learning BASIC for the PET gives excellent grounding for BASIC in most other computers.

### ***CALCULATOR MODE***

One of the features of the PET is its ability to be used as a calculator—a very sophisticated one. In general, calculator or immediate mode is in use whenever a BASIC statement is typed without a statement number. For instance, type the statement:

```
PRINT 5*2.7
```

and press return. This statement is in calculator mode and the answer

```
13.5
```

is displayed immediately. If, however, you type the statement

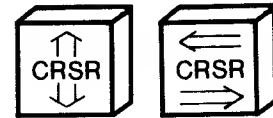
```
10 PRINT 5*2.7
```

it is in program mode since a statement number (10) is used. This statement is stored directly in the PET's memory instead of immediately calculating the answer. Now type

```
? 14/2
```

and press return. Did the answer 7 appear? The question mark is an abbreviation for the command PRINT and is a much more convenient form of the command. Try a few more calculations of your own

to get the feel of the PET's capability. Don't worry about making mistakes or doing something that might damage the machine. Nothing you type in can damage it.

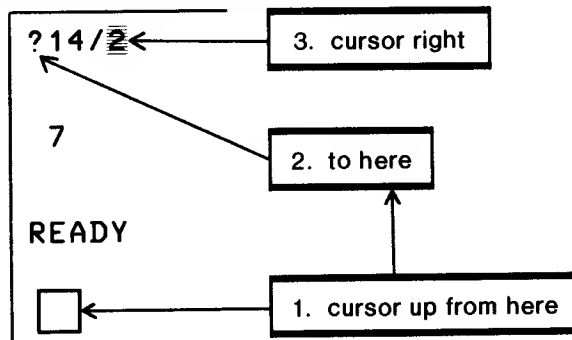


## HOW TO USE CURSOR CONTROLS

Do you recall the cursor controls at the top of the numeric pad? These are the arrows that point up, down, left and right. Now re-enter the command

? 14 / 2

and press return to get the answer. Next use the up cursor control to move back to the command and the right cursor to move to the 2. Like this!



Notice that the cursor moves over the characters without changing them in any way. Now type a 5 on top of the 2. Now when you press Return the answer given is 2.8 replacing the previous answer.

Try

? 245 \* 598

The answer should be 146510



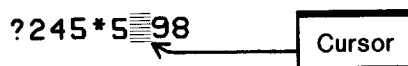
## INSERT/DELETE

The Insert/Delete keys are situated at the upper right corner of the number pad. These keys allow you to make space in a line either to insert characters that were missed or to delete characters that should not be there.

Move the cursor up to the 9 of 598 from the previous exercise. The screen looks like this:



Hold the shift key and simultaneously press INST once. Now the screen shows



leaving a space between the 5 and the 9. Type a decimal point(.) so you get

? 245 \* 5 . 98

and then press Return. The answer 1465.1 replaces the previous result.

To use the Delete, the cursor is moved to the character to the right of the one you wish to delete and the DEL key is pressed. Try this to change the 5.98 to 5.8 and then press return to get your answer.

Did you move the cursor to the 8? If so, the answer you received should be 1421.1.

Cursor controls can be used to correct any entry on a line. For instance, you might try typing a sentence and then make changes to it using the cursor controls with the insert and delete.



## ***CLEAR/HOME***

This key is located at the upper left corner of the number pad. Pressing the HOME key causes the cursor to move immediately to the upper left corner of the screen, called the Home position. Pressing shift and CLR together clears the screen and homes the cursor. This is sometimes useful to remove the displayed results from one program before going on to the next one. Clearing the screen does not clear program memory but only the screen itself. Memory may be cleared using the NEW command.

Enter the BASIC code:

```
10 PRINT 10/5+35
```

when you press Return no answer is given since this is a BASIC statement in program mode, not immediate mode.

Hold shift and press CLR clearing the screen. Now type the command LIST and notice what happens. The PET displays the command you previously entered. Statement 10 is still available since it was stored in memory. Type the command RUN and the program is run displaying the result 37.



## ***RUN/STOP***

Occasionally you may get into a program that has not provided a way to get out. Well-written programs will always provide a way to terminate the program when you are finished. But sometimes this provision was not made. In that case you can exit from a program by pressing STOP. The PET confirms your instruction by displaying

```
READY.
```

followed by the flashing cursor.

Now try this program.

```
10 PRINT I
20 I=I+1
30 GO TO 10
```

Type RUN and after a few seconds press STOP. This will bring the program to a screeching halt. In rare cases a programmer may have disabled the STOP. If this happens the only alternative you have is to turn off the power and then turn it on again. Since this action erases program memory it should be used only as a last resort. Hopefully, the program was saved on tape so it can be loaded once again.

The RUN key (shifted) lets you load and run a program from the disk. This is equivalent to typing a DLOAD and RUN except the RUN key does not permit the entry of a program name.



## **BASIC NUMBERS**

A large percentage of programs are number-oriented, and those that are not usually need some number manipulation for their operation. BASIC uses two types of numbers: integers and real (floating point). Another type of number uses scientific notation, which is simply a variation of the real number. Numbers may be used in programs for calculations, comparisons, as data in DATA statements, or as input from the keyboard. If theory doesn't interest you, you need only scan the following pages briefly for now.

### **Integers**

An integer is a number with no fractional part, thus no decimal point. The integer may have a sign for negative (−) or positive (+) numbers. Some examples of integers are:

1  
12  
−2358  
−25  
+31278

### **INTEGERS**

Integers have a maximum range from −32767 to +32767. An integer uses only 2 bytes of memory and can therefore save a lot of memory space in large programs. Why 2 bytes? Well, a byte occupies 8 bits or binary digits. Two bytes then have 2x8 or 16 bits available for use. One of these bits is used for a sign, leaving 15 for storing numbers. Now each binary digit can store two possible values; 1 or 0. With 15 bits available this allows a total of 2 to the power of 15 numbers, which is 32768. If one of these numbers is reserved for the value zero this leaves 32767 as the maximum. Makes sense doesn't it?

Integers may also be represented as real numbers.

### **Real Numbers**

A real number can be an integer, a fractional number, or a combination of both. Real numbers, like integers, may also be signed. For example:

1.  
1.5  
276.075  
.0125  
−123456789  
−.0000001

### **REAL NUMBERS**

A real number in the PET uses five bytes of memory and can have eight or sometimes nine digits not counting the sign or decimal point.

If excessive digits are to the right of the decimal point, roundoff occurs. Try entering

? .9876543216  
.987654322

### **ROUNDING**

The 6 is truncated and the last digit, 1, rounds up to 2.

? .9876543211  
.987654321

In this case the last 1 is truncated but the remaining 1 is not rounded since the last digit was less than 5.

```
? .9876543214
.987654322
```

## TRUNCATING

This number rounds incorrectly due to the conversion to floating point done internally in the PET's memory.

Real numbers are useful for the majority of applications on the PET whether for business, education, or the home. Real numbers can represent dollars, quantities, marks, account numbers, and so forth.

Now try this interesting exercise: Light travels at a speed of 186,200 miles per second. How many miles does it travel in a minute? In an hour? Day? Year? The distance light travels in a year is called a light year. So using the PET as a calculator try these calculations. Using the cursor to change the Print command each time will make this easier.

Distance light travels in a minute

```
? 186200*60
11172000
```

Distance light travels in an hour

```
? 186200*60*60
670320000
```

Distance light travels in a day

```
? 186200*60*60*24
1.608768E+10 ←
```

what happened  
here?

Distance light travels in a year

```
? 186200*60*60*24*365
5.8720032E+12 ←
```

and here?

Real numbers can also be small fractional values. For instance, computers do things in very small fractions of a second. These measurements are in milliseconds (1/1000 of a second), microseconds (1/1,000,000), nanoseconds (1/1,000,000,000), and the really fast ones in picoseconds (1/1,000,000,000,000). PETs basically operate in microseconds. Now let's find out how far light or electricity travels in these small periods of time.

Distance light travels in a millisecond

```
? 186200/1000
186.2
```

Distance light travels in a microsecond

```
? 186200/1000000
.1862
```

Distance light travels in a nanosecond

? 186200 / 100000000  
1.862E-04 ←

more strange  
results

Distance light travels in a picosecond

? 186200 / 1000000000000  
1.862E-07 ←

Did you know that the nearest galaxy to Earth is the Andromeda Galaxy? It is 2,300,000 light years away. Can you use your PET to find out how many miles this is?

Now let's find out about those weird numbers we got in some of the previous calculations.

### Scientific Notation

Real numbers may also be represented in scientific notation if they exceed their defined maximum size. The result of the calculation for the number of miles light travels in one day exceeded the maximum size for a real number so the PET automatically converted it to scientific notation. The nanosecond result was smaller than the smallest real number so it was converted to a small fractional number in scientific notation. Try typing a 1 followed by nine zeros.

? 1000000000  
1E+09

The result is a number (1E+09) in scientific notation that represents the original number in powers of ten. This number may be thought of as

$1 \times 10^9$

Some valid scientific notations are:

Scientific Notation	Actual Value
2.7385E8	273850000
1.085215E-7	.000001085215
-45E12	-45000000000000
-21E-15	-.000000000000021

The maximum ranges for floating point numbers in scientific notation are

largest       $\pm 1.70141183E+38$

smallest      $\pm 2.93873588E-39$

Now we can understand the previous results. The number of miles light travels in a light year was 5.8720032E+12, which is 5872003200000 miles and the picosecond distance of

1.862E-07, which is .0000001862 miles.

How many Feet? Inches? Centimeters? Millimeters?

Do you know about the Googol? It is a 1 followed by 100 zeros.

## STRINGS

A string permits BASIC programs to manipulate data that is not used for arithmetic calculations. Strings are enclosed in double quotes (") and may contain any letter, number, special character, graphic symbol, or cursor control character. Some strings are

```
"SPACE INVADERS"  
"DIFFICULTY LEVEL 1 to 9"  
"-----"  
"Want to try again?(Y/N)"
```

**STRINGS**

Strings may be up to 255 characters in length, which is more than adequate for most applications.

## VARIABLES

A variable is a name that you give to a memory location that is capable of storing a value. It might help to think of memory as consisting of a number of small boxes (figure 3.1). Every time you need a place to store a number or name you put a label on one of the boxes and place your value in the box. Suppose your program requires values to represent speed and complexity. Then one box could be given the variable name S (speed) and the other C (complexity). To place the numbers in the box (memory) you use an assignment statement.

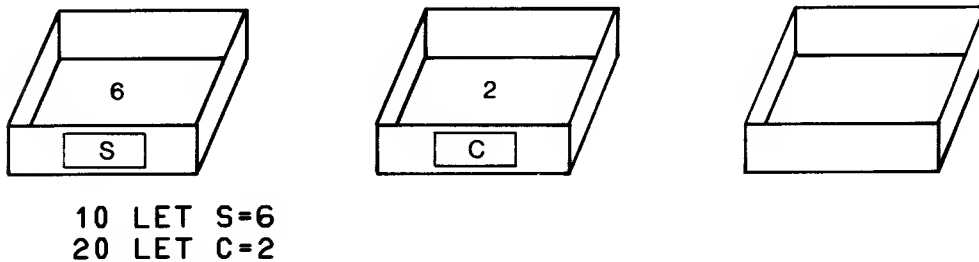


Figure 3.1 Use of variables

Statements 10 and 20 are assignments which give location S a value of 6 and C a value of 2. PET also lets you do this without the keyword LET as follows:

```
10 S=6  
20 C=2
```

Variable names for real numbers may be a single letter, a letter followed by a number, or two letters. Applying these rules gives the following valid names.

A  
A5  
T  
SM  
X1

### VARIABLE NAMES

Integers may be defined by placing a percent sign (%) after the variable name.

A%  
D4%  
HP%  
C%

### INTEGER NAMES

Variable names for strings also follow the above rules except that a string name ends with a dollar sign(\$). Thus some valid string names are:

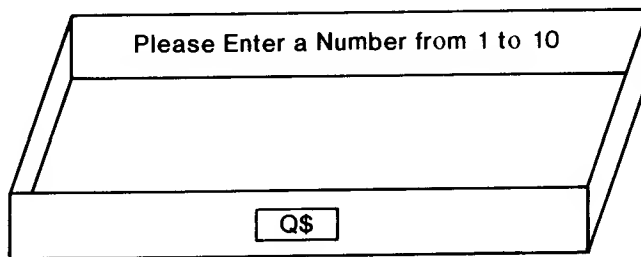
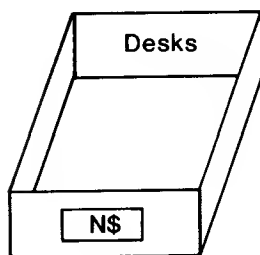
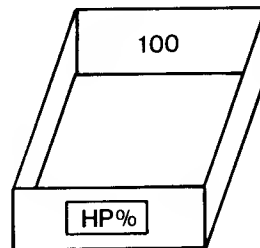
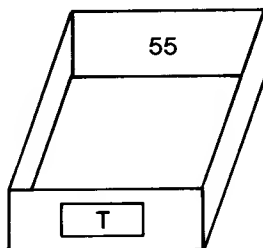
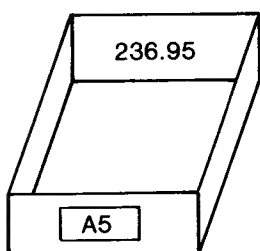
N\$  
AD\$  
R9\$  
Q\$

## **STRING NAMES**

Some valid assignments for these names are:

10 A5=236.95  
20 T=55  
30 HP%=100  
40 N\$="DESKS"  
50 Q\$="PLEASE ENTER A NUMBER FROM 1 TO 10"

## **ASSIGNMENTS**



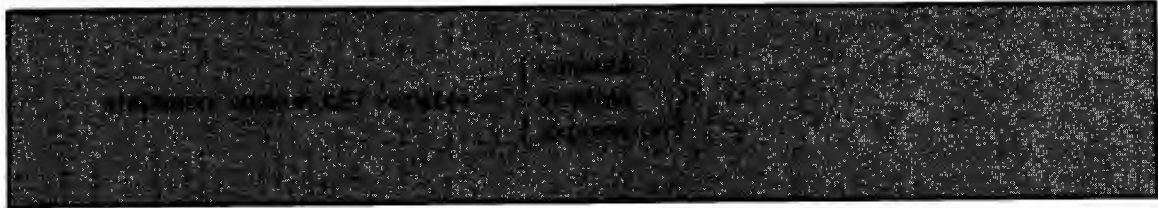
When choosing variables names care should be taken to avoid selecting names that are also reserved words. This problem is particularly prevalent with two-letter variables. When you are unsure of a word, a quick look at Appendix C can verify if your choice is a reserved word or not.

## **MULTIPLE STATEMENTS PER LINE**

The PET has a useful feature that permits you to enter several commands on one line. This feature may be used when several small steps are required and it seems unnecessary to type several lines. This feature also saves storage and may be useful to conserve space in longer programs. To place several statements on a line simply separate each statement with a colon (:).

10 N=1:K=25:X=0

## ARITHMETIC STATEMENTS



We have already seen the use of the LET for assigning values to a variable such as

```
10 LET N=25
```

The 10 in this statement is called a statement number. Each statement in BASIC requires a unique statement number except when multiple statements per line are defined. Statement numbers increase in value through the program. The value of the increment is up to the programmer but a value of 10 or more is advised since this will leave space to insert additional statements if they are eventually required.

The LET statement also permits the calculation of values using the common arithmetic operations as well as a number of mathematical functions. In every case the calculation is specified on the right of the equal sign and the variable to which the result is assigned is on the left of the equal sign.

### Add (+)

Addition is the same as it is in the immediate mode, except for the statement number. The plus sign specifies the addition of two numeric values, which may be integer, real, or scientific notation.

10 N=25	Result	
20 K=14.5		
30 J=N+K	J is 39.5	
40 N=N+1	N is 26	ADD
50 K=K+.25	K is 14.75	
60 L=J+20	L is 59.5	
70 L=L+N+K	L is 100.25	

### Subtract (—)

Subtraction is handled in the same way as addition. The only real difference is that the result can sometimes be a negative number. In this case the PET automatically retains the sign on the result.

10 M=40	Result	
20 J=55.5		
30 F=J-M	F is 15.5	SUBTRACT
40 G=M-J	G is -15.5	
50 H=J-M-20	H is -4.5	

### Multiply (\*)

To multiply the PET uses the asterisk symbol. All the usual rules in mathematics apply.

10 A=12	Result	
20 B=5.3		
30 C=A*B	C is 63.6	MULTIPLY
40 D=A*-5	D is -60	
50 E=A*B*D	E is -3816	

### Divide (/)

Division follows normal rules of mathematics with the dividend on the left of the slash and the divisor to the right of the symbol.

	Result	
10 W=48		
20 X=12		
30 Y=W/X	Y is 4	DIVIDE
40 Z=W/-4.8	Z is -10	
50 Z=Y/W/.25	Z is 1	

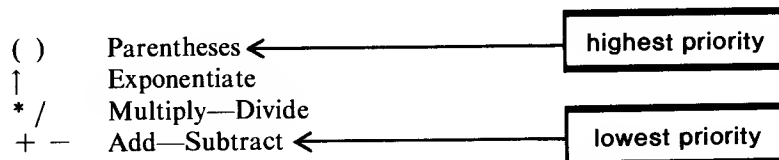
### Exponentiation (^)

Exponentiation is simply raising a number to the power of a second number. This is most common when we square a number or cube it, multiplying the number by itself two or three times respectively. But the PET can raise a number to any power, even fractional or negative powers.

	Result	
10 P=5		
20 Q=25		
30 R=P^2	R is 25	POWERS
40 S=Q^3	S is 15625	
50 T=P^1.6	T is 13.132639	
60 U=Q^-2.5	U is 3.2E-04	

### HIERARCHY AND PARENTHESES

Most arithmetic operations are straightforward but what happens when there is a mixture of different operations in one statement? When any of the five operators are mixed, BASIC applies a rule of hierarchy to the expression. This rule defines a certain priority to the operators and defines which goes first.



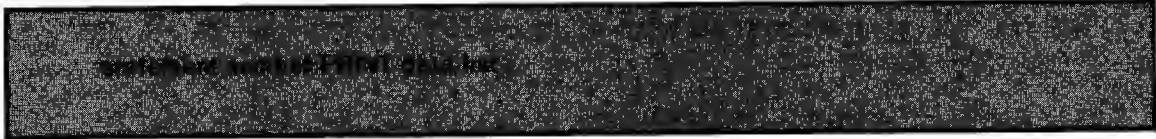
Normally arithmetic operations proceed left to right but when operators are mixed the rules of priority take effect. For instance, the expression

10 N=8-4\*3

must be evaluated with the multiply operation first. Therefore the correct answer to this calculation is -4, not 12 (which would be the case if the PET evaluated from left to right). To change the order parentheses may be used. Since parentheses have the highest priority, an operation within the brackets will be done first.

10 N=(8-4)*3	
RUN	
12 ←	answer

**PRINT**



The **PRINT** statement is used to display information on the screen. Usually the Print displays one line at a time from left to right, 40 characters per line, and from the top to bottom of the screen. When the bottom line (line 25) is reached all lines are scrolled up by one line and the first line disappears from the top of the screen. The format of the print is the keyword followed by a list of variables, constants, or expressions separated by commas or semicolons.

```
100 PRINT A,B
200 PRINT N,25
300 PRINT L,"SUM OF A AND B",A+B
```

When the parameters are separated by commas each value is displayed in a separate ten position zone on the screen. The PET therefore has four zones on one line and can have one value in each of these zones.

Diagram illustrating a 4-column layout structure. The columns are labeled 1, 10, 20, 30, and 40. The layout is divided into four zones:

- ZONE 1 (Columns 1 to 10)
- ZONE 2 (Columns 10 to 20)
- ZONE 3 (Columns 20 to 30)
- ZONE 3 (Columns 30 to 40)

Now try this program:

```
10 A=-2.5
20 B=30
30 PRINT A,B,A+B
RUN ←
```

**don't forget**

**-2.5                      30**

27.5 ←

**gives this**



Each of the three values is allocated ten spaces. The first of these spaces is either blank or, in the case of a negative number, shows the sign. If a PRINT has more than four values to print they will overflow to the next line.

Alphanumeric strings are also given a ten-character zone in which to print.  
Now try this program:

```
10 PRINT "TWO", "STRINGS"
20 PRINT "ONE LONG STRING", "SHORT ONE"
RUN
```

TWO            STRINGS ← gives  
ONE LONG STRING            SHORT ONE

Since strings have no sign the first character prints in position 1 of the zone. In some cases, such as statement 20, a string may exceed the length of a zone. When this happens the string continues into the second zone without interruption. A subsequent string or value will start in the next available zone.

Closer spacing of values may be achieved with the use of the semicolon between values. When a semicolon is used instead of a comma for numeric values one space is left between each value plus a second space for the sign if the number is negative. Alphabetic fields have no space when a semicolon is used. Example:

```
10 PRINT 12;24;-7
20 PRINT "SUM OF A + B";12+24
30 PRINT "TWO";"STRINGS"
RUN
```

12 24 -7  
SUM OF A + B 36 ← gives  
TWO STRINGS

Some additional useful features of the Print are the use of a single Print statement to leave a blank line. This is a command like

```
10 PRINT ← prints empty line
```

with no variables listed. A second useful feature, which will be considered in more depth later, is the ability to place cursor controls in a string. For instance, the statement:

```
10 PRINT "[clr]" ← this character      "♥" ← looks like this
```

where [clr] represents the shifted CLR control, which causes the screen to be cleared and the cursor homed before subsequent printing is done. This command is great for clearing extraneous characters from the screen and gives the following output a clear and uncluttered place to display.

Try this program:

```
10 PRINT "THIS CLUTTERS THE SCREEN"
20 PRINT "CLUTTER"
30 PRINT "CLUTTER"
40 PRINT "CLUTTER"
50 FOR I=1 TO 1000:NEXT I
60 PRINT "[clr]"
70 PRINT "ON A CLEAR SCREEN YOU CAN SEE YOUR OUTPUT"
RUN
```

slows things down  
press shift clear

## INPUT



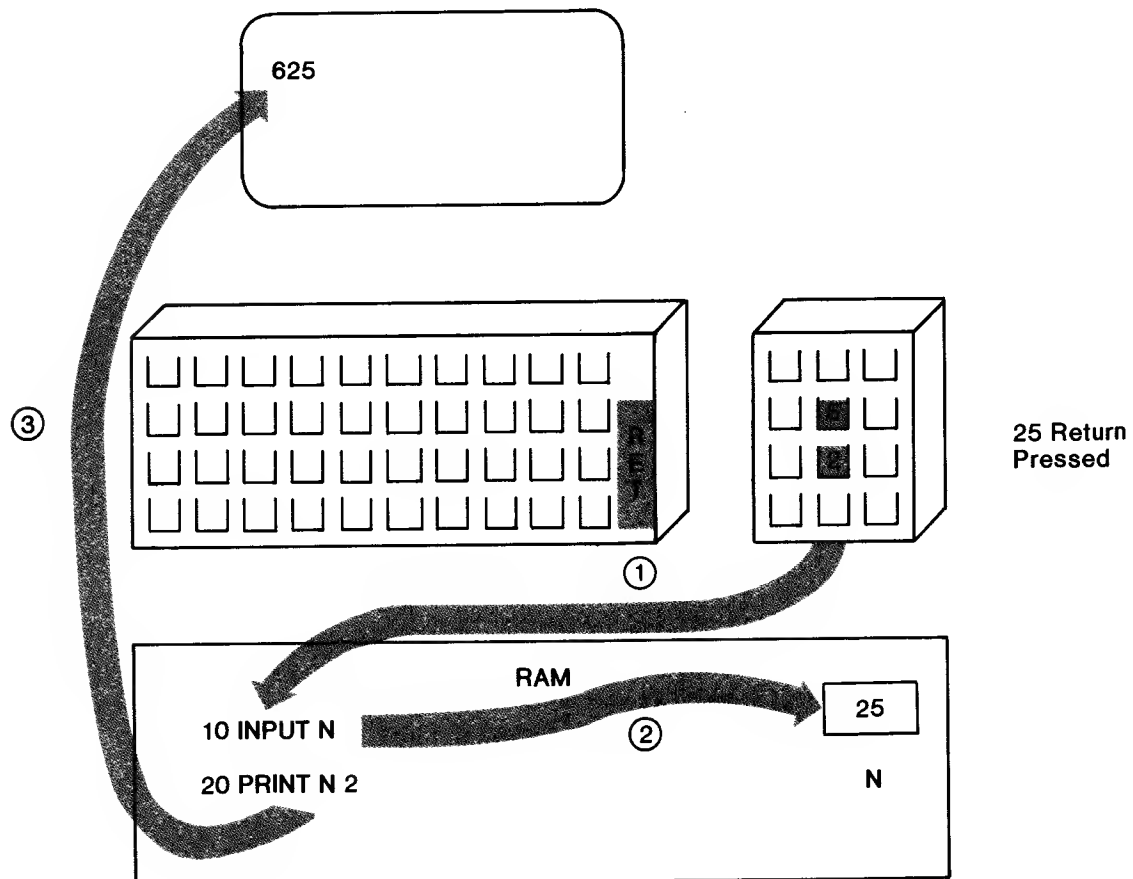
The INPUT statement is used to let the user of the program supply data for processing by the program. When an INPUT is encountered during program execution the computer waits for the user to type in a value or values and then press Return. When Return is pressed the values are entered into the program's variables and the program continues with these values. For example,

```
10 INPUT N
20 PRINT N^2
RUN
?25
625
```

value entered

result printed

When this program is RUN a question mark appears indicating the program is waiting for a value for N. If 25 is typed and Return pressed, the value 25 is now contained in the variable N and is used for the calculation in statement 20.



More than one value may be entered by separating them with commas. For instance:

```
10 INPUT A,B
20 PRINT A*B
```

expects two values to be entered, one for A and the second for B. These values must be entered with a comma between them as follows:

? 12,25

One problem with the above examples is that the program's user may not know what is expected as input when the program is run. This subject is considered in depth in the chapter "Interacting with the User of Your Program," but for now it is sufficient to introduce the use of a character string in the INPUT statement. In the first example we could have written

```
10 INPUT "ENTER A VALUE FOR N";N
20 PRINT N
RUN
```

ENTER A VALUE FOR N ? 937.5

937.5

cursor stops here

input N

print N

Notice the use of the semicolon in statement 10 to separate the string from the variable N. The semicolon is necessary to avoid a syntax error and it also lets the user type N's value directly following the message which asks for it.

## GOTO

statement number GOTO statement number

A GO TO is a branching statement that causes the program to change the flow of execution and continue at the statement number identified in the GO TO. GO TO may be written with or without the space. The statement

```
100 GOTO 20
```

causes the program to branch from statement 100, where it is currently, to statement 20 where the program continues. This statement is called an unconditional branch since branching occurs regardless of what has been happening up to this point in the program.

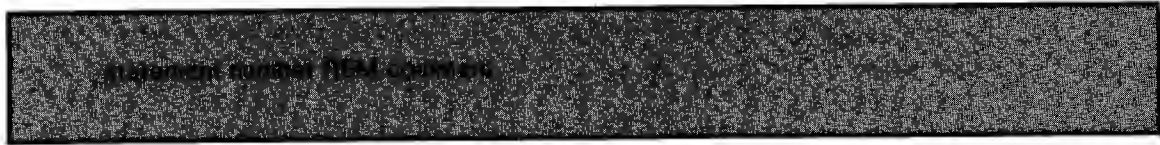
```
10 INPUT "STARTING VALUE, INCREMENT";S,I
20 PRINT S
30 S=S+I
40 GOTO 20
```

to here

branches from here

This program will accept a starting value and a value by which S is to be increased. The program proceeds to print S and then increases it by the increment. Statement 40 then branches to 20 where the new value is printed and the process continues. Since a GO TO is unconditional this program will run forever (well, not quite, since eventually S will run out of space and an illegal quantity error will terminate the program).

## **REM**



The REMark statement simply makes life easier for you as a programmer. It has no effect on your program but prints whatever it includes within the program. The REM is useful for including comments to yourself or anyone else who may read your program. Ideally these comments should make the program more readable and understandable to anyone who reads it. Don't make the mistake of thinking that since you understand what you have written, you don't need to use REMs. It's surprising how much you will forget after a few days or weeks.

```
10 REM DISPLAYS SCORE
40 REM CALCULATES TRAJECTORY
```

## **SIMPLE CALCULATION PROGRAM**

For our first program let's try something fairly simple. Suppose we want a program that takes two numbers and adds, subtracts, multiplies, and divides them and then prints the results.

Since this is a relatively small program we might be successful by simply sitting down at the PET and writing it. However, to develop good program design habits we should engage in some planning before actually writing the program. A very effective tool for program design is called English code or Pseudo code. This approach involves writing down a few basic steps that constitute a correct general solution to the problem. For instance, we might write:

1. Input Values
2. Do Calculations
3. Print Results

These steps are called the top level of our solution. If the problem is more complex then these steps, it may be broken down into smaller parts. Step 1 could break down to:

1. Input Values
  - 1.1 Input First Value
  - 1.2 Input Second Value

Step 2 could break down into:

2. Do Calculations
  - 2.1 Calculate Sum
  - 2.2 Calculate Difference
  - 2.3 Calculate Product
  - 2.4 Calculate Quotient

Now if we combine all this English code the solution looks like figure 3.2.

This may seem like a lot of work to prepare for a simple program but as programs that you write become longer and more complex this approach will help you to be organized and produce much more successful programs. More on this later.

1. Input Values
  - 1.1 Input First Value
  - 1.2 Input Second Value
2. Do Calculations
  - 2.1 Calculate Sum
  - 2.2 Calculate Difference
  - 2.3 Calculate Product
  - 2.4 Calculate Quotient
3. Print Results

**Figure 3.2** Simple calculation program English code

Now by following the English code we can write the PET BASIC program in figure 3.3. Statements 110 and 120 use INPUT to accept values for A and B, the first and second values. These values are then used in lines 140 to 170 to find the sum, difference, product, and quotient of A and B. Finally, statements 190 to 220 print the answers. For good measure 230 goes back to 110 for a new set of values. We'll discuss looping in English code in later programs. Pay close attention to the method used to print the results such as:

```
190 PRINT A;"+";B;"=";S
```

The semicolons are used to concatenate on the output line the values for A and B with the symbols + and = followed by the calculated result S. The effect of this statement is an output that looks like an arithmetic expression.

```
100 REM SIMPLE ARITHMETIC CALCULATIONS
110 INPUT "ENTER FIRST VALUE";A
120 INPUT "ENTER SECOND VALUE";B
130 REM DO CALCULATIONS +,-,*,/
140 S=A+B
150 D=A-B
160 P=A*B
170 Q=A/B
180 REM PRINT RESULTS
190 PRINT A;"+";B;"=";S
200 PRINT A;"-";B;"=";D
210 PRINT A;"*";B;"=";P
220 PRINT A;"/";B;"=";Q
230 GOTO 110
```

**Figure 3.3** Simple calculation program

### **FAHRENHEIT—CELSIUS PROGRAM**

This program is a bit more practical but is also quite easily developed. We would like to input either Fahrenheit or Celsius degrees and have the computer give us the alternative temperature. Since we haven't yet learned decision-making statements for the PET we will write this as two separate programs combined into one.

The first will be program 100 and will convert Celsius to Fahrenheit. Here is the English code.

#### Program 100

1. Input Celsius
2. Calculate Fahrenheit
3. Print Results

The second is program 200, which converts Fahrenheit to Celsius. Here is its English code.

#### Program 200

1. Input Fahrenheit
2. Calculate Celsius
3. Print Results

The program is shown in figure 3.4. The first part is in statements 100 to 150, which converts Celsius to Fahrenheit degrees and the second part from 200 to 250 converts Fahrenheit to Celsius. To run part one you may enter either

**RUN**

or

**RUN 100**

Either command starts the program at statement 100. The program then asks for a Celsius degree, which it then converts to Fahrenheit. This process continues as long as you like. To stop it press STOP and then Return or simply press Return without entering a temperature.

Now to convert the other way enter

**RUN 200**

which starts the program at statement 200 and proceeds to do the other conversion.

```
100 REM CONVERT CELSIUS TO FAHRENHEIT
110 INPUT"ENTER CELSIUS DEGREES";C
120 F=C*9/5+32
130 PRINT C;"DEGREES CELSIUS IS";F;"FAHRENHEIT"
140 PRINT
150 GOTO 110
200 REM CONVERT FAHRENHEIT TO CELSIUS"
210 INPUT "ENTER FAHRENHEIT DEGREES";F
220 C=(F-32)*5/9
230 PRINT F;"DEGREES FAHRENHEIT IS";C;"CELSIUS"
240 PRINT
250 GOTO 210
```

**Figure 3.4** Fahrenheit—Celsius program

### **IMPROVING YOUR SOLUTION**

When you have used this program for a while it becomes apparent that some improvements could be made. One problem is for conversions to Celsius, which often come out as long fractional values. If you look up the INT function in the index you will discover a way to improve this output, which is implemented in figure 3.5 in line 220.

```

100 REM CONVERT CELSIUS TO FAHRENHEIT
105 PRINT "[clr]"
110 INPUT "ENTER CELSIUS DEGREES";C
120 F=C*9/5+32
130 PRINT C;"DEGREES CELSIUS IS";F;"FAHRENHEIT"
140 PRINT
150 GOTO 110
200 REM CONVERT FAHRENHEIT TO CELSIUS"
205 PRINT "[clr]"
210 INPUT "ENTER FAHRENHEIT DEGREES";F
220 C=INT((F-32)*5/9)
230 PRINT F;"DEGREES FAHRENHEIT IS";C;"CELSIUS"
240 PRINT
250 GOTO 210

```

**Figure 3.5** Improved Fahrenheit—Celsius program

INT is a language feature called a function that extracts the integer part of the number or variable contained within its brackets. For example the expression `INT(22.7777778)` gives the result of 22. Normally the number is in a variable and appears as follows in the INT function: `INT(C)`. If in this example we wanted to round the result giving 23 instead of the unrounded answer of 22 the expression `INT(C + 0.5)` could be used. An arithmetic expression may also be used in the function as shown in figure 3.5.

```

RUN 200
ENTER FAHRENHEIT DEGREES ? 73
73 DEGREES FAHRENHEIT IS 22.7777778 CELSIUS

```

this would look better  
if INT were used

```

73 DEGREES FAHRENHEIT IS 22 CELSIUS

```

A second irritation is that whenever you change from part one of the program to part two, as described earlier, the output from previous operations remains on the screen. This output is cleared in statements 105 and 205, which use the CLR control character in the character string of the PRINT statement as follows:

```

105 PRINT "[clr]"

```

### **WEIGHTED AVERAGE PROGRAM**

A common need in education is to apply different weights to grades received depending upon the importance of a particular project. Grades themselves may also be given different marking schemes so that determining a student's final mark becomes a difficult task. This program will make a start at solving this problem, although in a somewhat limited way.

To develop any program we need to have a clear mental picture of what we are attempting to accomplish. In the previous programs this was easy, but real life is not always this easy. For the weighted average problem a diagram of what we are attempting to do will help us to get a grasp of the problem. This is what the picture looks like:

Test	1	2	3	4	5
Maximum Marks	30	25	50	10	75
Weight Factor	2	1	2	3	2

This information gives us some idea about the types of processing we will need to do in our program. In addition to this we need to define the input that is expected and what output is required. The following illustrates our requirements for input and output.

ENTER FIVE MARKS SEPARATED BY COMMAS  
FOR EXAMPLE:

25, 15, 20, 38, 7, 70 ←

input

? 27, 20, 38, 7, 70

MARK	MAXIMUM	WEIGHT	PERCENT
27	30	2	18
20	25	1	8
38	50	2	15
7	10	3	21
70	75	2	18

output

TOTAL WEIGHTED AVERAGE PERCENT 80 ←

ENTER FIVE MARKS SEPARATED BY COMMAS ←  
FOR EXAMPLE: 25, 15, 40, 7, 63

output preparation  
for next grades

Once we have designed the required input and output and considered the processing requirements we are ready to develop the English code. Be careful at this stage of development of trying to do too much. Keep it simple and quite general. In other words do an overview.

1. Set up Initial Values
2. Input Five Marks
3. Calculate Percents
4. Print Results
5. Calculate Total WA Percent

As usual, each step of the English code, as necessary, is developed further. Step 1 sets up each of the maximum values and the weights.

1. Set up Initial Values
  - 1.1 Clear Screen
  - 1.2 Set Maximum Marks
  - 1.3 Set Weights
  - 1.4 Find Total Weight



Next we will do step 3 since 2 requires no further elaboration.

3. Calculate Percents

3.1 Find Percent for each of the 5 Marks (Relative to 100%)

Notice that it is not necessary at this time to fully define each statement for the program. We are basically creating an outline to assist in writing the program once we are satisfied with our solution. Now we are ready to print the results.

4. Print Results

4.1 Print Heading

4.2 Print each of 5 Marks and Related Values

The last step is to calculate the total weighted average percent and print it in step 5.

5. Calculate Total WA Percent

5.1 Find Total Percent

5.2 Print Total

Now we are ready to write the program, which is shown in figure 3.6.

First the program assigns maximums for each test in variables M1–M5. This indicates test 1 had a maximum of 30 marks, test 2 of 25, and so on. Next, line 140 allocates the weights for each test. Weight 2 for test 1, weight 1 for test 2, and so on to weight 2 for test 5.

```
100 REM WEIGHTED AVERAGE
105 PRINT "[clr]"
110 REM MAXIMUM MARK FOR EACH TEST
120 M1=30:M2=25:M3=50:M4=10:M5=75
130 REM WEIGHTING FOR EACH TEST
140 W1=2: W2=1: W3=2: W4=3:W5=2
150 TW=W1+W2+W3+W4+W5
160 PRINT "ENTER FIVE MARKS SEPARATED BY COMMAS"
170 PRINT "FOR EXAMPLE:"
180 PRINT "25,15,40,7,63"
190 PRINT
200 INPUT MA,MB,MC,MD,ME
210 P1=INT((MA/M1*100)*W1/TW)
220 P2=INT((MB/M2*100)*W2/TW)
230 P3=INT((MC/M3*100)*W3/TW)
240 P4=INT((MD/M4*100)*W4/TW)
250 P5=INT((ME/M5*100)*W5/TW)
260 PRINT "[clr]MARK","MAXIMUM","WEIGHT","PERCENT"
270 PRINT
280 PRINT MA,M1,W1,P1
290 PRINT MB,M2,W2,P2
300 PRINT MC,M3,W3,P3
310 PRINT MD,M4,W4,P4
320 PRINT ME,M5,W5,P5
330 PT=P1+P2+P3+P4+P5
340 PRINT
350 PRINT "TOTAL WEIGHTED AVERAGE PERCENT";PT
360 GOTO 160
```

Figure 3.6 Weighted average program

Statements 160 to 200 accept the marks for one student and then 210 to 250 calculate the percentage for each test by applying the appropriate weight. Lines 260 to 350 then print the results including an overall percentage for the student.

One limitation of this program is the need to do a separate calculation for each percent even though they are similar. Another limitation is the need for five Print statements for the output. Another problem will surface if marks are entered that exceed the maximum for that test. This program will cheerfully accept such a mark with no arguments. These problems will be considered in an improved program in the next chapter.

## COMPUTING LOAN PAYMENTS

Which of us doesn't need to take out a loan on occasion for a new car, an appliance, or vacation? With today's high interest rates a small difference in the percentage can make a large difference in monthly payments. We may also want to try different repayment terms to evaluate our ability to pay each month. Even with our limited knowledge of BASIC at this stage we can still write a useful program.

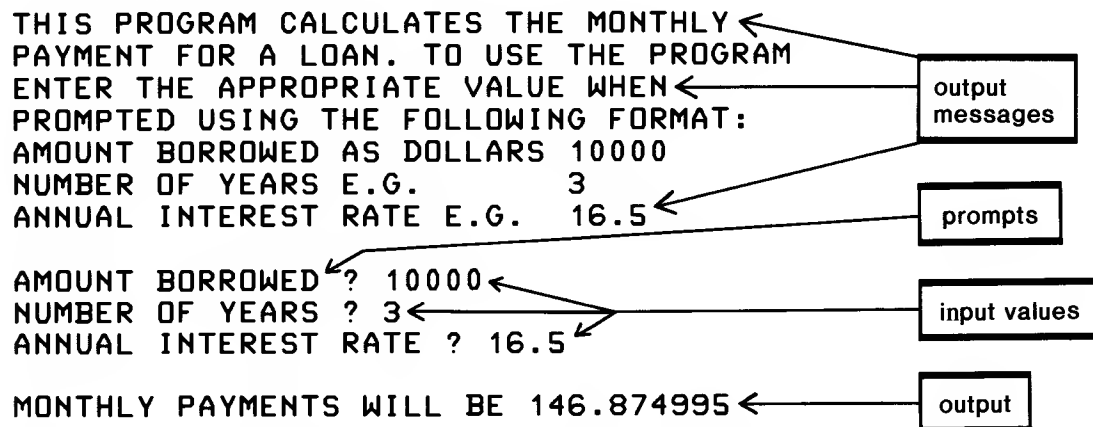
First we need a formula to calculate the payment given the other variables. The Periodic Rent of Annuity formula can be found in Business Math or Accounting books. It is as follows:

$$p = a \left( \frac{i}{1 - (1 + i)^{-n}} \right)$$

where:

a is the amount borrowed  
i is the interest rate  
n is the number of periods  
p is the payment

The approach taken in this program is to initially print some instructions about the program's use. Included in these instructions are some sample data to clarify what the program expects from the user. This method is particularly important here since an entry such as interest rate could be entered several different ways such as .165, 16.5, or 16.5%, but only 16.5 would give the correct results. Here is a sample of what we want the input and output to look like:



Since the formula needs the interest as a fraction for the period of the loan, we will need to divide the input by 100, giving the fraction, and then divide again by 12 to convert to a monthly rate. Since personal loans are normally taken out for annual periods (one, two, three years, and so on) that is how the term will be read, although a two-and-one-half year loan could be entered as 2.5. We will then convert this value into months, which is necessary for the formula.

Now we are prepared to develop the English code.

1. Print Instruction Messages
2. Input Amounts Required
  - 2.1 Input Amount Borrowed
  - 2.2 Input Number of Years
  - 2.3 Input Interest Rate
3. Convert to Internal Amounts
  - 3.1 Multiply Years  $\times$  12 Giving Months
  - 3.2 Divide Interest/100/12 (Converts to Monthly Fraction)
4. Calculate Payment
5. Print Payment

To reiterate, the principle in problem solving is to define in general terms your solution and then to break down these components until you have completely solved the problem. By following each step defined above, the program is written.

```
100 REM CALCULATING MONTHLY LOAN PAYMENTS
110 PRINT "THIS PROGRAM CALCULATES THE MONTHLY"
120 PRINT "PAYMENT FOR A LOAN. TO USE THE PROGRAM"
130 PRINT "ENTER THE APPROPRIATE VALUE WHEN "
140 PRINT "PROMPTED USING THE FOLLOWING FORMAT:"
150 PRINT "AMOUNT BORROWED AS DOLLARS 10000"
160 PRINT "NUMBER OF YEARS E.G.      3"
170 PRINT "ANNUAL INTEREST RATE E.G.  16.5"
180 PRINT
190 INPUT "AMOUNT BORROWED";A
200 INPUT "NUMBER OF YEARS";N
210 INPUT "ANNUAL INTEREST RATE";I
220 N=N*12      :REM CONVERT TO MONTHS
230 I=(I/100)/12 :REM % PER MONTH
240 P=A*(I/(1-(1+I)-N))
250 PRINT
260 PRINT "MONTHLY PAYMENTS WILL BE";P
```

### REVIEW QUESTIONS—CHAPTER 3

1. What is meant by calculator or immediate mode?
2. How are the four cursor controls used? Explain how to correct a line such as ?435\*18 that should have been a divide (/) instead of multiply (\*), when the cursor is on the first position of the next line.
3. Explain how to correct the above command when the 435 should have been 4.35 and the cursor is already on the same line.
4. Describe the purpose of the CLR/HOME key.
5. Discuss two kinds of numbers used on the PET/CBM.
6. Explain the difference between rounding and truncating.
7. What is a number such as 5.8720032E+12 called? What is the equivalent decimal value?
8. What is a variable? Why is it used in programming? What are the rules for valid variable names?
9. Explain how data may be assigned to a variable.

10. Write BASIC statements for the following algebraic expressions ( $\pi = 3.1415$ ).

a)  $h = a \cdot b$

b)  $a = b + c - d - e$

c)  $c = \frac{a \cdot b}{e}$

d)  $x = \frac{y + z}{w \cdot v}$

e)  $j = \frac{k + (i/n)}{m}$

f)  $a = \pi r^2$

g)  $v = \frac{4}{3} \pi r^3$

h)  $a = \pi r \sqrt{r^2 + h^2}$

i)  $e = a^{a^j}$

j)  $k = \sqrt{\frac{x + y}{z}}$

11. Explain the difference between using commas to separate variables in a PRINT statement and using semicolons.
12. How can you clear the screen from within a BASIC program?
13. What is the purpose of the INPUT statement? Describe what happens when a statement like 10 INPUT "RADIUS";R is used in a program.

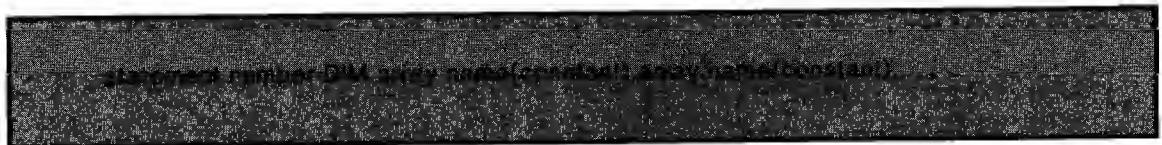


# 4

## ***Not So Basic BASIC***

**T**he previous chapter introduced some of the elements of the BASIC language; enough of them to let us write some fairly useful programs. But to really take advantage of the PET/CBM's capability it is necessary to understand statements that permit decision making, looping, data structuring in arrays, and the use of subroutines for program organization and efficiency. These additional features of the language will broaden our horizons considerably and give us the ability to solve a wider range of interesting problems.

### ***DIM***



The DIM or Dimension statement permits the storage in our program of multiple data values under the same variable name. For example, in the last chapter we wrote a program to do weighted averages which used five variables M1 to M5 for maximum marks, five more W1 to W5 for the weight, and MA to ME for the actual marks. Using DIM these 15 variables could be reduced to 3 or, with a little ingenuity, to a single variable.

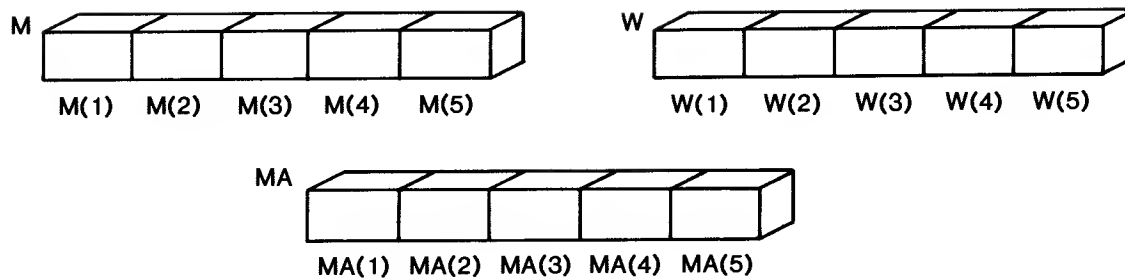
Variables defined with DIM are called arrays and may each have a number of elements or positions in which to store data. The variables for weighted average could be described in BASIC as follows:

```
10 DIM M(5),W(5),MA(5)
```

The number 5 in parentheses defines the number of elements in each array. Each element of the array may be referenced by using a subscript with the array name. For instance, position 3 of the array for maximum marks would be referenced in the program by specifying M(3).

Arrays are particularly useful because the subscript can also be a variable or an expression. This feature permits array references such as

```
50 PRINT MA(I)  
60 P=MA(N)/M(N)*100  
70 H=W(K-1)
```



**Figure 4.1** Visualizing a one-dimensional array

The arrays that we specified in statement 10 can be visualized as shown in figure 4.1. Actually the PET allocates 0 as an element, so there are really six elements in each array. For convenience we will usually ignore the zero element, although at times it may be useful.

### Storing Values in an Array

If we wished to store 10 numbers in an array, an array of 10 elements would be defined by a DIM statement. The following program sets up such an array, reads 10 numbers into the array and then displays the contents of the array. The 10 numbers are entered in real time from the keyboard, one value at a time.

```

10 DIM K(10)
20 REM LOAD ARRAY WITH NUMBERS
30 I=1
40 INPUT "NUMBER ";K(I)
50 I=I+1
60 IF I<=10 THEN 40
70 REM PRINT THE NUMBERS
80 I=1
90 PRINT "NUMBER ";I;" = ";K(I)
100 I=I+1
110 IF I<=10 THEN 90
120 PRINT "DONE"
130 END

```

IF I IS LESS THAN OR  
EQUAL TO 10 THEN  
GO TO STATEMENT 40

Otherwise  
come here

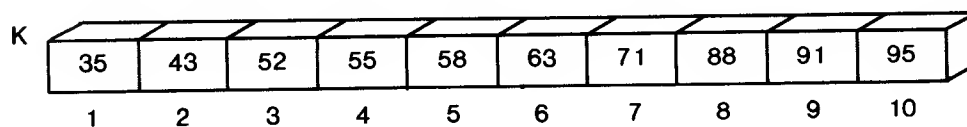
When this program is run the user is prompted for each of the 10 numbers to be entered into the array. These entries would appear on the screen as follows:

```

NUMBER ? 35
NUMBER ? 43
NUMBER ? 52
NUMBER ? 55
NUMBER ? 58
NUMBER ? 63
NUMBER ? 71
NUMBER ? 88
NUMBER ? 91
NUMBER ? 95

```

When the program reaches statement 70 the contents of array K will appear as follows:



Statements 70 to 120 now display these contents to confirm that the numbers are actually in the array. Here is the output.

```
NUMBER 1 = 35
NUMBER 2 = 43
NUMBER 3 = 52
NUMBER 4 = 55
NUMBER 5 = 58
NUMBER 6 = 63
NUMBER 7 = 71
NUMBER 8 = 88
NUMBER 9 = 91
NUMBER 10 = 95
DONE
```

Arrays with one dimension such as we have used here are the most common type used in BASIC programs and we will have occasion to use them frequently in other programs in this and subsequent chapters. But first let's look at arrays that have two or more dimensions.

### **Multi-Dimensional Arrays**

Arrays may be multi-dimensional although more than two or three dimensions are rarely useful. Specifications in BASIC such as:

```
10 DIM A(5,4),B(25,3)
```

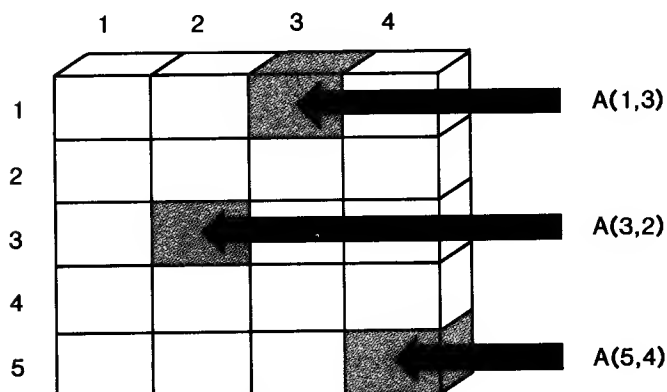
are two-dimensional arrays while

```
10 DIM C(3,5,4),D(2,5,20)
```

are three-dimensional.

For example, the first array defined above ( A(5,4) ) refers to a two-dimensional array with five elements by four elements. This array can be visualized as a rectangular shape containing five rows and four columns as shown in figure 4.2. Of course, in the PET or CBM's memory the array does not actually exist in this physical shape but thinking of a two-dimensional array in this way helps us write programs that need this kind of array.

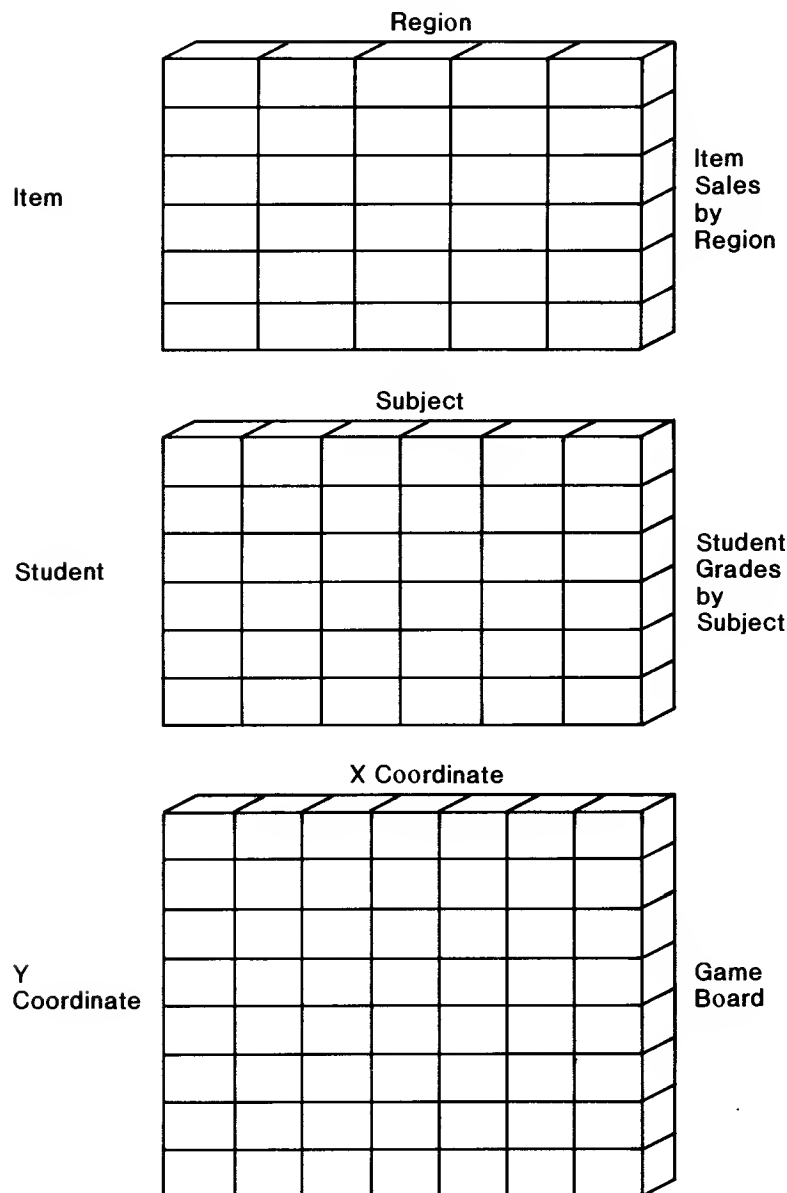
A



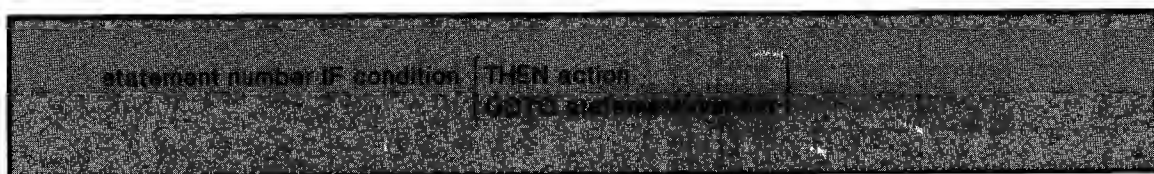
**Figure 4.2** A two-dimensional array A(5,4)



Two-dimensional arrays can be used to represent many types of data. Here are just a few examples.



### IF—THEN



The GOTO was an unconditional branch but when it is used with an IF statement conditional branching may be used. The IF also permits the programmer to selectively do one or more operations depending upon circumstances in the program. Generally the IF statement examines a condition and if the condition evaluates true it does the action specified. If the condition is false the program merely continues at the next statement. Figure 4.3 identifies the operators available for decision making.

Operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or Equal to
<=	Less than or Equal to
<>	Not Equal to

**Figure 4.3** Logical operators

Some possible IF statements are:

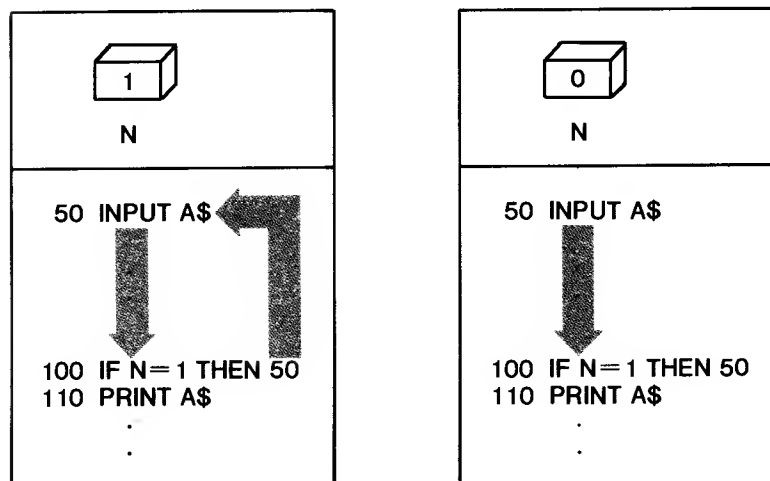
```

100 IF N=1 THEN 50
200 IF K>245 GOTO 50
300 IF R$="YES" THEN 5000
400 IF W(I)<=M(K) THEN 100

```

Notice that the THEN or GOTO may be used interchangeably and can cause branching either forward or backward within the program. All numeric and alphanumeric variables may be compared in the IF statement although *numeric variables* may not be compared to alphanumeric variables.

In line 100 above, if the value of N is 1 when statement 100 is reached then branching will go to statement 50 where the program will continue. If N is any other value than 1 then the program would continue at the next statement following statement 100.



Actions do not have to be GOTO's. They may be almost any other BASIC statement. Note the following examples:

```

100 IF K=L THEN PRINT K,J
200 IF N$="JONES" THEN C=C+1
300 IF Q=1 THEN INPUT "PLEASE ENTER NAME";N$

```

An additional feature of PET BASIC is the ability to specify more than one action to be taken when a condition is true. Each action in the IF statement is separated from the other with a colon(:).

```

100 IF N = 100 THEN N=N+1:PRINT S:GOTO 10
200 IF M$(J)="JULY" THEN A$=M$(J):T=T+A:K=0

```

Using this feature can reduce a lot of unnecessary branching thus making your programs easier to follow. Statements written this way also execute faster than if they were individual statements with individual line numbers. A possible disadvantage to this use of multiple statements is that program changes and maintenance can become more difficult.

Note the use of M\$(J) in statement 200. This example indicates that a string variable may also be used as an array. The only difference between a numeric array and a string array is that the contents of the string array will be string data.

The operators AND and OR (figure 4.4) may also be used to combine several conditions in one IF statement. For instance, if you wanted to branch to statement 20 either when a count had reached 100 or when a code of 3 was found in an array D, the following statement could be used.

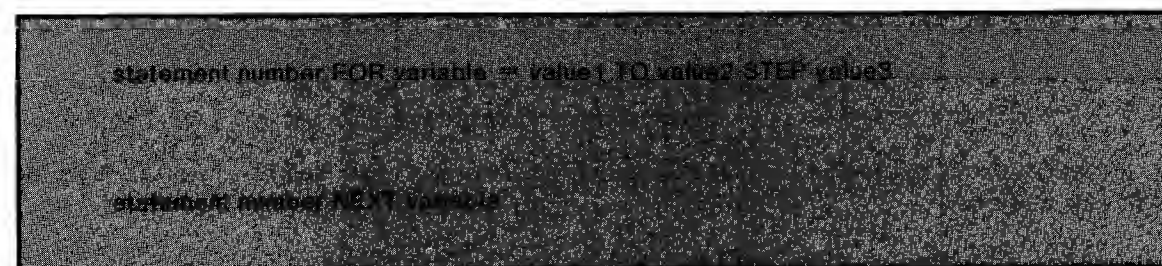
**200 IF C=100 OR D(I)=3 THEN 20**

The OR evaluates true when either one condition or the other or both are true. AND evaluates true only when both conditions are true.

IF condition-1 { AND OR } condition-2 THEN action			
condition-1	condition-2	Resulting Boolean Operator Value	
		AND	OR
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

**Figure 4.4** Boolean operators

## FOR—NEXT



The FOR and NEXT statements are used to provide for looping in a program. Although looping may be accomplished with a combination of arithmetic, IF and GOTO statements the FOR—NEXT provides a simpler way of achieving the same thing.

For example if we wanted to print out the numbers from 1 to 10 we could write the program:

```
10 N=0
20 N=N+1
30 PRINT N
40 IF N<10 GOTO 20
```

Output

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

Using the FOR—NEXT the previous program looks as follows:

```
10 FOR N=1 TO 10 STEP 1
20 PRINT N
30 NEXT N
```

Output

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

end of loop adds 1 to N  
and checks if 10 is reached

The FOR statement defines the variable to be used (N) for the loop, its starting and ending values, and the amount to increment (STEP) N when the end of the loop (NEXT) is reached. Any number of statements may appear inside the loop, including another FOR—NEXT pair. When the increment is 1, as in this case, the STEP may be deleted giving the same effect.

```
10 FOR N=1 TO 10
```

Variables used by the FOR must be numeric but can be fractional if required. The following statements will print the fractional values from 1.5 to 2.2 across one line of the screen.

```
10 FOR N=1.5 TO 2.2 STEP 0.1
20 PRINT N;
30 NEXT N
```

increment of 0.1

semicolon overrides  
the print margin

Output

1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2

The FOR may also use negative values for its operation. Often negative amounts may appear in the STEP value although they could be used for starting or ending values as well.

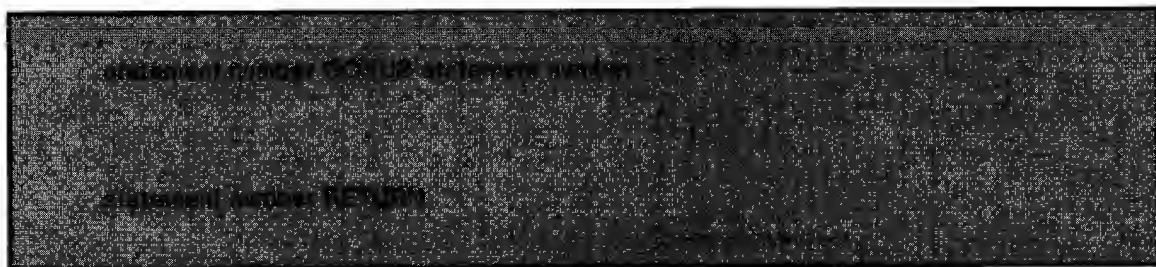
```
10 FOR I=10 TO 1 STEP -1
20 PRINT I
30 NEXT I
```

negative  
increment

Output

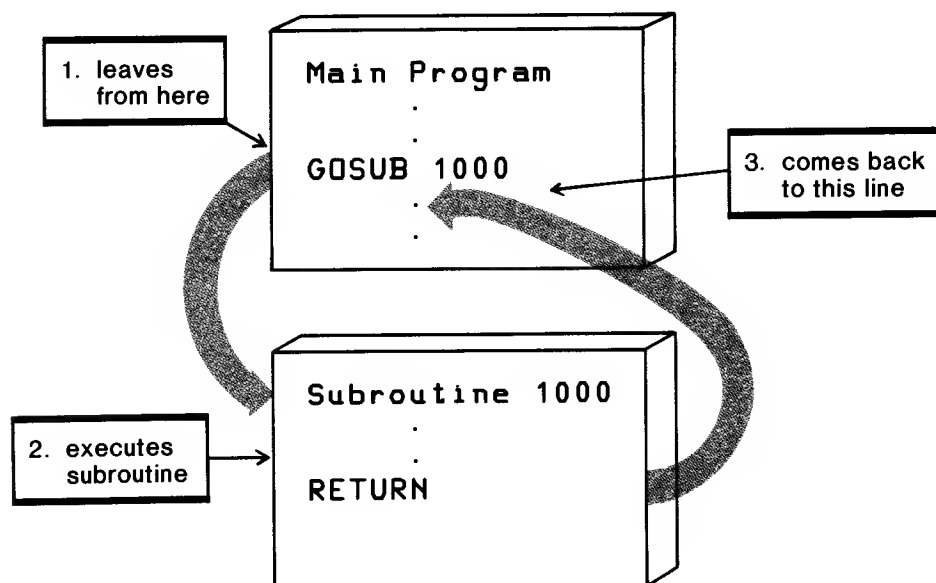
10  
9  
8  
7  
6  
5  
4  
3  
2  
1

### GOSUB—RETURN

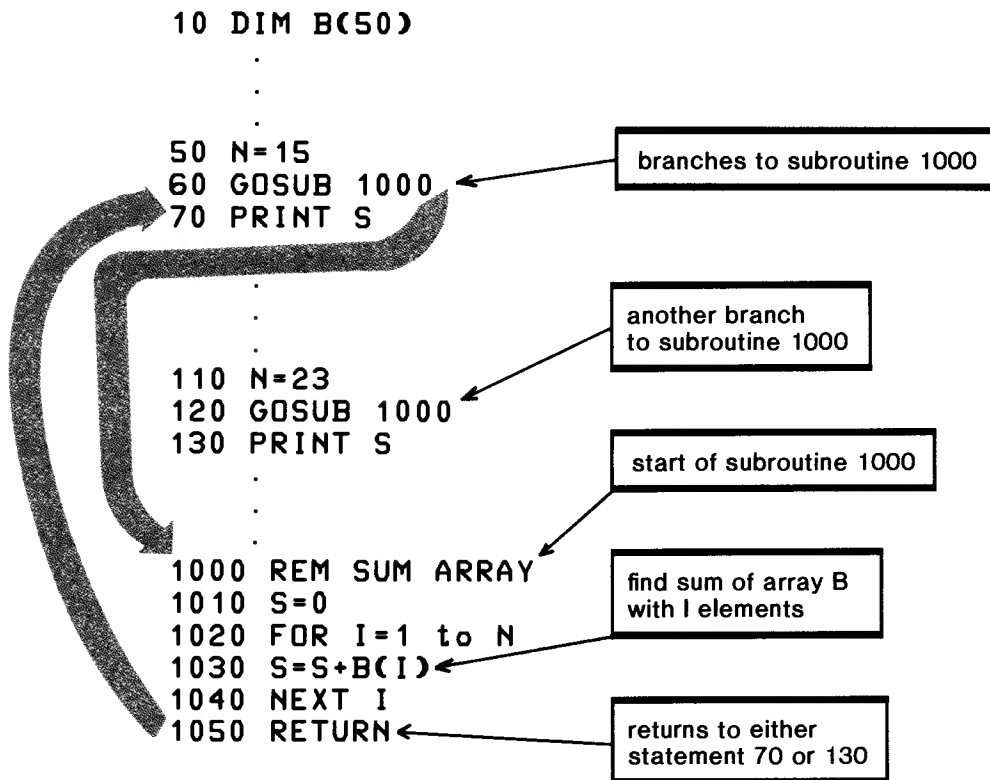


GOSUB acts much like the GOTO except the PET remembers where it came from and can then RETURN back to the statement following the GOSUB in the program. GOSUB means go to a subroutine that begins at the statement number specified in the GOSUB itself. A subroutine is a set of BASIC statements written to perform a specific action. For instance, a subroutine may be written to accept the input from a user, to plot a graph on the screen, or to check for errors after a disk operation.

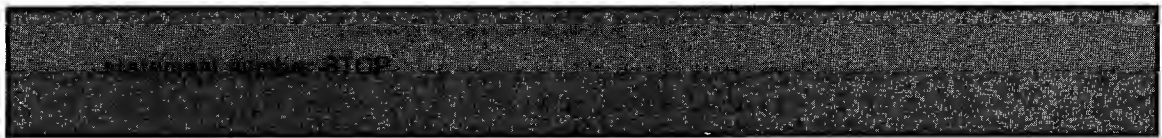
At the end of the subroutine a RETURN statement tells the computer to revert to the location following the GOSUB in the main program. A nice thing about GOSUB is it may go to a subroutine from many different places in the same program. This is useful since program code that is common to different parts of the program may be written once as a subroutine but used as often as is needed.



Suppose subroutine 1000 is written to find the sum of values previously stored in an array. The variable N defines how many values are in the array at any one time and the sum is created in S by the subroutine itself.



## STOP



The STOP command may be used to terminate a program when the program has finished its processing. STOP may be used as an independent statement as in

```
200 STOP
```

which terminates after a sequence of program steps have been completed or it may be used in a decision

```
150 IF A$="NO" THEN STOP
```

for examining a user's response to a question such as "DO YOU WANT TO CONTINUE", as in the following sequence of statements

```

140 INPUT "DO YOU WANT TO CONTINUE";A$
150 IF A$="NO" THEN STOP
160 GOTO 10
  
```

## GENERATING RANDOM NUMBERS

Some subroutines, known as functions, are already built into the PET/CBM's ROM and can be directly accessed by the program. The first of these we will use is the RND function which is needed to generate random numbers. Normally RND produces fractional random numbers such as .974626516 and not integers.

Try these statements on your system:

```
? RND(1)
.974626516
```

```
? RND(1)
.747932106
```

```
? RND(TI)
.435687146
```

The resulting numbers given here will be different from yours since a random number is given each time. The use of TI uses the value of the timer to initiate action for the random number (TI is described fully in chapter 6).

In addition to being fractional these numbers may not be within the range required for a specific problem. In the game which follows, for example, we want only whole numbers in the range of 1 to 100. To limit the range of generated numbers to numbers no greater than 100 we can multiply the random number by 100.

```
? RND(TI)*100
```

If the random number had been .974626516 this expression would transform the number into the value 97.4626516 since multiplying has shifted the decimal point 2 places to the right. Now to convert to an integer the INT function comes in handy.

```
? INT(RND(TI)*100)
```

RND function nested  
in the INT function

The expression we now have will generate the random number and take only the integer part of it thus giving us 97 or some other 2 digit integer. At this point, if we try a number of values we find that we have created values from 0 to 99 rather than 1 to 100; so we simply add 1 to the result, which gives the values required for our game. A statement like

```
710 N=INT(RND(TI)*100)+1
```

will be used in the program to generate a random 2-digit number from 1 to 100 and store that number in the variable N.

The number we started with would now become 98 and be the first random number to be stored in N. The next might be .428313721 which would result in the number 43 in variable N.

## **NUMBER GUESSING GAME**

Part of the enjoyment derived from having a PET or CBM computer is playing the numerous games available. This availability should not discourage you from developing your own games, such as the following example of how a game program may be written. Although the program is quite short, for a game, it does have many of the characteristics a good game should have. These qualities are user instructions, interaction with the player, a certain degree of randomness, and the capability to play the game repeatedly if the user desires. Figure 4.5 shows the English code.

### **Number Guessing Game Mainline**

1. Play game until done
  - 1.1 Print Instructions Subroutine
  - 1.2 Generate Number Subroutine
  - 1.3 Accept Guess Subroutine

### **Print Instructions Subroutine**

1. If Instructions not wanted then  
Return
2. Display Instructions
3. Return

### **Generate Number Subroutine**

1. Generate random number
2. Print prompt
3. Return

### **Accept Guess Subroutine**

1. Accept guesses until correct answer
  - 1.1 Accept guess
  - 1.2 Check answer
  - 1.3 If wrong give a hint
2. Display number of tries
3. Set number of guesses to 0
4. Return

**Figure 4.5** Number guessing game English code

This solution shows how to use subroutines effectively for good program organization. These subroutines print user instructions, generate the random number, and accept the player's guess. The program statements from 100 to 180 (figure 4.6) control these subroutines and determine if the player wishes another game.

Several new features of BASIC are used in the program in addition to variations on features already discussed up to this point.



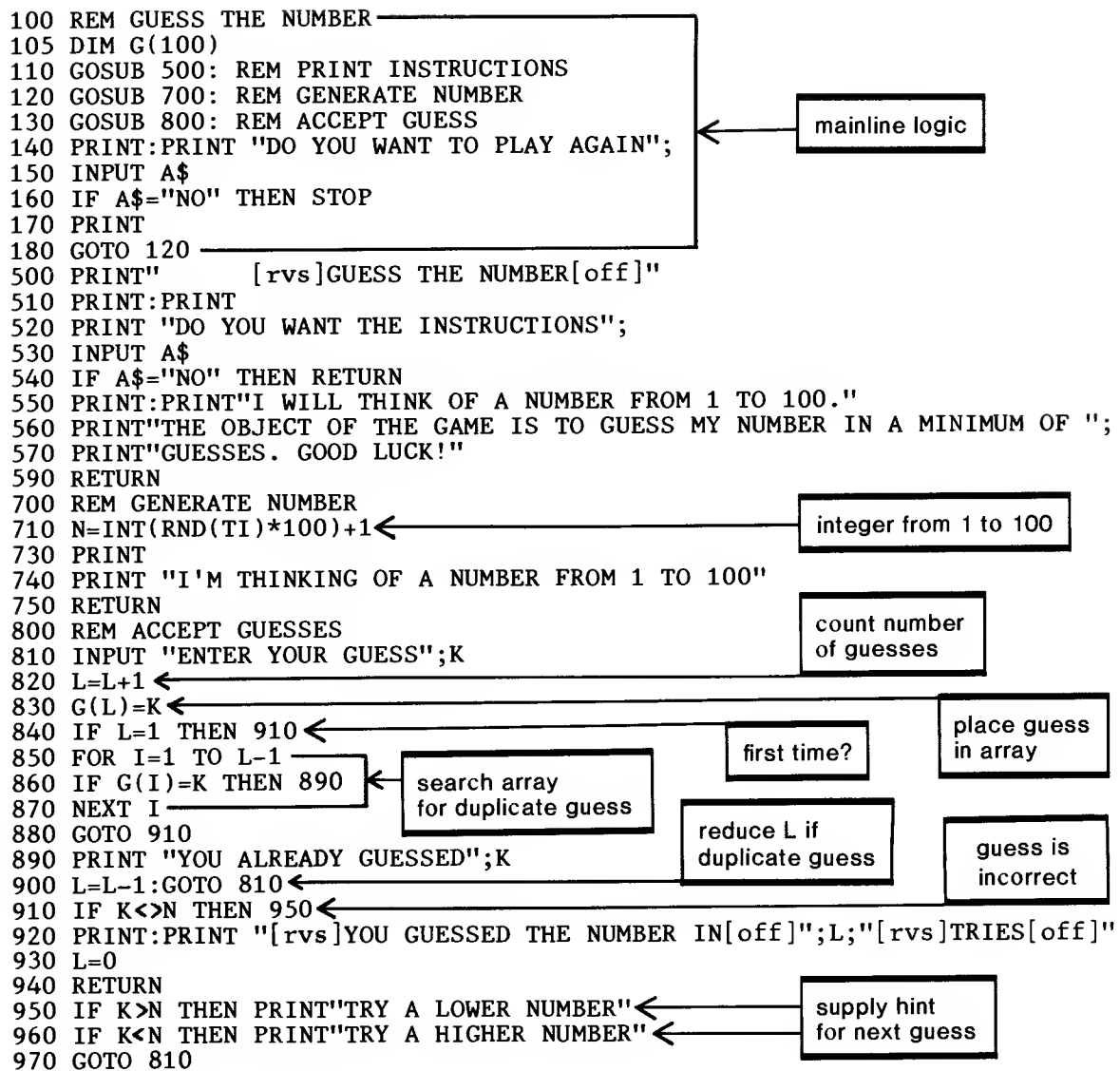


Figure 4.6 Number guessing game

## TIME DELAYS

After you have played the number guessing game a few times you may wish the computer wasn't so fast. In fact the speed at which the computer operates can be a psychological disadvantage since the user will feel a constant pressure to perform even when there is no need to hurry.

The use of time delay program code to slow down the action of the computer while the player takes some action is a useful tool in this program. The delay is created by including a FOR-NEXT loop that simply takes up some processing time before moving on to the next statement. The larger the value in the loop the longer the delay. This use of the FOR-NEXT shows the optional use of the NEXT statement without the variable I. The loop

```
580 FOR I=1 TO 3000:NEXT
```

will occupy about three seconds of computer time. The purpose in this program is to give the user some time to read the instructions before being rushed into guessing a number. Although the program could be left without a delay there is a psychological benefit to be derived by not creating undue pressure on the user. Of course in some situations time pressure may be an integral part of the game.

Try these two delays in the previous program and see if they don't improve the operation considerably.

```
580 FOR I=1 TO 3000:NEXT
720 FOR I=1 TO 500:NEXT
```

Statement 580 gives a three-second delay after the instructions have been displayed. If three seconds is too short then a larger value like 5000 or more could be tried.

Statement 720 gives the appearance of the computer taking its time to determine a random number. Although the user will need to wait during this delay it is only about half a second. This has the effect of making the action more comfortable without actually creating a noticeable delay. Figure 4.7 contains the revised program with the time delays included.

```
100 REM GUESS THE NUMBER
105 DIM G(100)
110 GOSUB 500: REM PRINT INSTRUCTIONS
120 GOSUB 700: REM GENERATE NUMBER
130 GOSUB 800: REM ACCEPT GUESS
140 PRINT:PRINT "DO YOU WANT TO PLAY AGAIN";
150 INPUT A$
160 IF A$="NO" THEN STOP
170 PRINT
180 GOTO 120
500 PRINT"      [rvs]GUESS THE NUMBER[off]"
510 PRINT:PRINT
520 PRINT "DO YOU WANT THE INSTRUCTIONS";
530 INPUT A$
540 IF A$="NO" THEN RETURN
550 PRINT:PRINT"I WILL THINK OF A NUMBER FROM 1 TO 100."
560 PRINT"THE OBJECT OF THE GAME IS TO GUESS MY NUMBER IN A MINIMUM OF ";
570 PRINT"GUESSES. GOOD LUCK!"
580 FOR I=1 TO 3000:NEXT: REM TIME DELAY
590 RETURN
700 REM GENERATE NUMBER
710 N=INT(RND(TI)*100)+1
720 FOR I=1 TO 500:NEXT: REM TIME DELAY
730 PRINT
740 PRINT "I'M THINKING OF A NUMBER FROM 1 TO 100"
750 RETURN
800 REM ACCEPT GUESSES
810 INPUT "ENTER YOUR GUESS";K
820 L=L+1
830 G(L)=K
840 IF L=1 THEN 910
850 FOR I=1 TO L-1
860 IF G(I)=K THEN 890
870 NEXT I
880 GOTO 910
890 PRINT "YOU ALREADY GUESSED";K
900 L=L-1:GOTO 810
910 IF K<>N THEN 950
920 PRINT:PRINT "[rvs]YOU GUESSED THE NUMBER IN[off]";L;"[rvs]TRIES[off]"
930 L=0
940 RETURN
950 IF K>N THEN PRINT"TRY A LOWER NUMBER"
960 IF K<N THEN PRINT"TRY A HIGHER NUMBER"
970 GOTO 810
```

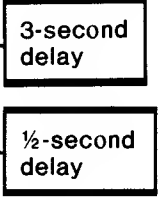


Figure 4.7 Revised number guessing game

## IMPROVED WEIGHTED AVERAGE PROGRAM

Now that we know about arrays let's take another look at the weighted average program from the previous chapter. Since there were always five marks to be processed with five weights and five maximums, each of these could become an array.

In addition to using arrays we would like to permit the user to enter marks for another student or terminate the program. This places most of the program logic into one large loop that is continually repeated until all students have been processed. The following English code implements the desired changes.

1. Set up Initial Values
  - 1.1 Set Maximum Marks
  - 1.2 Set Weights
  - 1.3 Find Total Weight
2. Do Weighted Average Until No More Students
  - 2.1 Input Five Marks
    - 2.1.1 Input Mark(i)
    - 2.1.2 Check for Maximum Allowed
  - 2.2 Calculate Percents and Print
    - 2.2.1 Find Percent for Each of the 5 Marks (Relative to 100%)
    - 2.2.2 Print Each of 5 Marks and Related Values
    - 2.2.3 Compute Total Weighted Percent
  - 2.3 Print Weighted Total
  - 2.4 Accept Another Student (Yes/No)?

```
100 REM WEIGHTED AVERAGE
110 DIM M(5),W(5),MA(5)
120 REM MAXIMUM MARK FOR EACH TEST
130 M(1)=30:M(2)=25:M(3)=50:M(4)=10:M(5)=75
140 REM WEIGHTING FOR EACH TEST
150 W(1)=2:W(2)=1:W(3)=2:W(4)=3:W(5)=2
160 FOR I=1 TO 5
170 TW=TW+W(I)
180 NEXT I
190 PRINT "[c1r]"
200 FOR I=1 TO 5
210 PRINT "ENTER MARK";I;"( MAX";M(I);")";
220 INPUT MA(I)
225 IF MA(I)>M(I) THEN PRINT "EXCEEDS MAXIMUM":GOTO 210
230 NEXT I
240 PRINT "[c1r]MARK","MAXIMUM","WEIGHT","PERCENT"
250 PRINT
260 REM COMPUTE PERCENTS AND TOTAL
270 FOR I=1 TO 5
280 P=INT((MA(I)/M(I)*100)*W(I)/TW)
290 PRINT MA(I),M(I),W(I),P
300 PT=PT+P
310 NEXT I
320 PRINT
330 PRINT "TOTAL WEIGHTED AVERAGE PERCENT";PT
340 PT=0
350 PRINT
360 PRINT
370 INPUT "TYPE (Y) FOR ANOTHER STUDENT";A$
380 IF A$="Y" THEN 190
390 STOP
```

set data  
in arrays

prompt

Figure 4.8 Improved weighted average program

The arrays are defined as M(5), W(5), MA(5) respectively. The program in figure 4.8 shows these arrays and the assignment of the maximum and the weight for each test. Now any reference to the arrays can be made using a FOR loop and a subscript.

A second improvement to the program prints the maximum mark allowed when asking for input. For instance, when the second mark is required the program prints:

**ENTER MARK 2 ( MAX 25 ) ?**

This request is printed from statement 210 where I provides the mark reference number and M(I) gives the maximum for the second mark.

A final improvement, unrelated to arrays, allows the user to enter data for another student or terminate the program, which is defined by the major loop at step 2 in the English code.

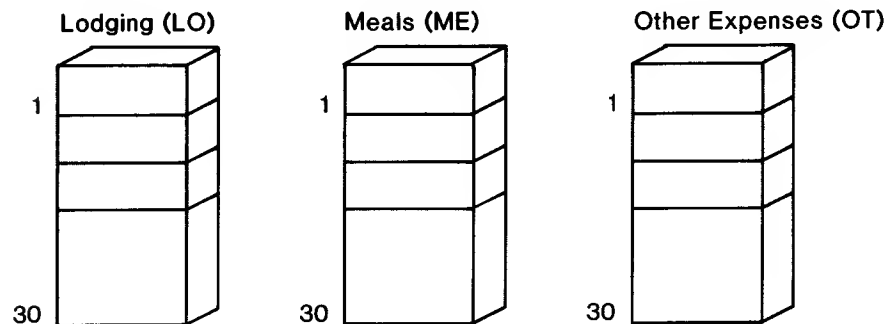
One additional advantage of arrays is the ease with which the program may be changed to accept more or even less data per student. This change is made by simply changing the DIM and each FOR loop. Of course the maximums and weights would also be changed accordingly. You might want to consider generalizing this program so that any number of marks could be entered depending upon the user's needs.

## ***CALCULATING TRIP COSTS***

How many times have you planned a trip and spent many hours calculating your anticipated expenses? This program assists in estimating trip costs assuming that you will be driving to and from your destination.

The principle is quite simple. Costs are based on gas costs for the entire trip plus other daily expenses such as lodging and meals. Once the basics of the program are understood modification is quite easy to provide for other personal needs you might have.

The program provides for up to 30 days' accumulation of data by using 3 arrays for Lodging (LO), Meals (ME), and Other Expenses (OT). These arrays as shown here define a maximum number of entries but our program will be designed to use only the portion of the arrays that is required.



Inputs to this program come in two categories. First we will need to know the costs for driving. To keep things simple this will be limited to gasoline costs but could easily be expanded to include other entries such as oil, tolls, repairs, and so on. To make things more interesting let's ask the user for the following inputs:

Initial Trip Inputs	
1.	Number of days on trip
2.	Total miles driven
3.	Average miles per gallon
4.	Average price per gallon

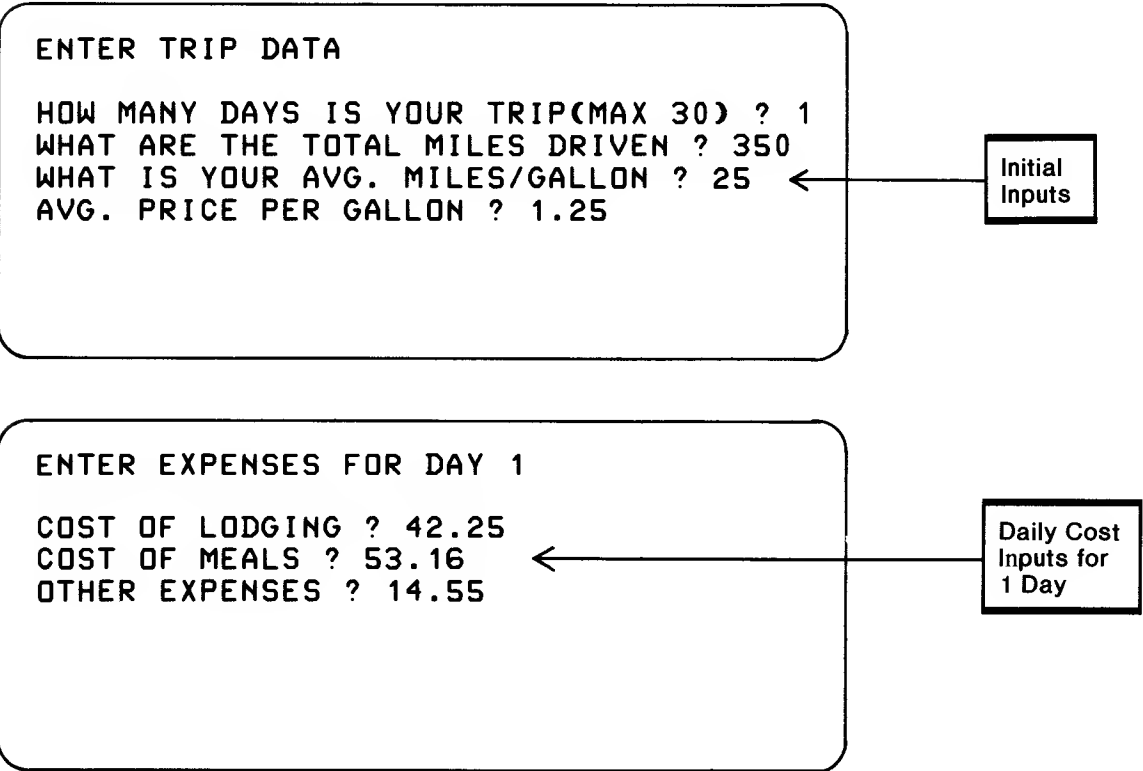
A second group of inputs are the daily expenses. Again there could be many entries here but we will keep this area quite simple while still allowing for additional entries later. Here are the daily inputs:

Daily Inputs
1. Cost of lodging
2. Cost of meals
3. Other expenses

The outputs required from this program would be an itemized list of the trip and daily costs and a total cost for the trip. As an additional output the average cost per day might be interesting, so let's include it in the output.

Trip Cost Outputs
1. Gasoline costs
2. Lodging
3. Meals
4. Other Expenses
5. Total trip cost
6. Average cost per day

Now that the input and output requirements have been defined in general terms let's develop a screen layout showing specific prompts, inputs, and outputs. This is easier now since we have at least outlined our needs above.



TOTAL TRIP COSTS		
GAS	17.50	
LODGING	42.25	
MEALS	53.16	
OTHER EXPENSES	14.55	
	-----	
TOTAL	127.46	
AVG COST/DAY	127.46	

←

Trip Cost  
Outputs

Now we are ready to develop the English code for the program logic. An overview of the solution suggests this will not be a lengthy program, nor will it be very complex, but rather consists of a lot of input and output operations.

#### Trip Costs English Code

1. Enter initial trip inputs
2. Enter daily costs
3. Calculate trip costs
4. Display trip costs

Now expand the code:

1. Enter initial trip inputs
  - 1.1 Accept days on trip
  - 1.2 Accept total miles driven
  - 1.3 Accept average miles per gallon
  - 1.4 Accept average price per gallon
2. Enter daily costs until last day
  - 2.1 Accept cost of lodging
  - 2.2 Accept cost of meals
  - 2.3 Accept other expenses
3. Calculate trip costs
  - 3.1 Calculate gas cost
  - 3.2 Sum daily costs until last day
    - 3.2.1 Accumulate lodging costs
    - 3.2.2 Accumulate meal costs
    - 3.2.3 Accumulate other costs
  - 3.3 Calculate total cost
  - 3.4 Calculate average cost per day
4. Display trip costs
  - 4.1 Display costs for gas, lodging, meals and other
  - 4.2 Display total cost
  - 4.3 Display average cost per day

English code lines 2 and 3.2 refer to repetition in the logic needed to control the activities on a daily basis. These lines are represented by FOR loops in the program in figure 4.9.

A slight improvement has been made when coding the program. That is a subroutine at 470 that clears the screen and prints down three lines. This has the effect of keeping the screen clear for each group of activities and moves the cursor down a few lines so the user is looking closer to the center of the screen rather than at the top all the time.

A second improvement is the use of the TAB feature in lines 390 to 450. TAB positions the cursor to the position indicated prior to printing the data. The TAB is discussed further in chapter 5.

```

100 REM CALCULATE TRIP COSTS
110 DIM LO(30),ME(30),OT(30)
120 GOSUB 470
130 PRINT "[rvs]ENTER TRIP DATA"
140 PRINT
150 INPUT "HOW MANY DAYS IS YOUR TRIP(MAX 30)";D
160 INPUT "WHAT ARE THE TOTAL MILES DRIVEN";M
170 INPUT "WHAT IS YOUR AVG. MILES/GALLON";A
180 INPUT "AVG. PRICE PER GALLON";C
190 FOR I=1 TO D
200 GOSUB 470
210 PRINT "[rvs]ENTER EXPENSES FOR DAY[off]";I
220 PRINT
230 INPUT "COST OF LODGING";LO(I)
240 INPUT "COST OF MEALS";ME(I)
250 INPUT "OTHER EXPENSES";OT(I)
260 NEXT I
270 REM CALCULATE COSTS
280 GC=M/A*C
290 FOR I=1 TO D
300 LC=LC+LO(I)
310 MC=MC+ME(I)
320 OC=OC+OT(I)
330 NEXT I
340 TC=GC+LC+MC+OC
350 AC=TC/D
360 GOSUB 470
370 PRINT "[rvs]TOTAL TRIP COSTS"
380 PRINT
390 PRINT "  GAS";TAB(18);GC
400 PRINT "  LODGING";TAB(18);LC
410 PRINT "  MEALS";TAB(18);MC
420 PRINT "  OTHER EXPENSES";TAB(18);OC
430 PRINT TAB(17);"-----"
440 PRINT "TOTAL";TAB(18);TC
450 PRINT "AVG COST/DAY";TAB(18);AC
460 STOP
470 PRINT "[clr]":PRINT:PRINT:PRINT
480 RETURN

```

**Figure 4.9** Calculating trip costs program

## REVIEW QUESTIONS—CHAPTER 4

1. Why are arrays important? In what situations should they be used?
2. Write DIM statements for each of the following:
  - a) a list of costs for 20 products.
  - b) storage for the names of the months, January to December.
  - c) a matrix of distances between 10 different cities.
3. Explain how data is read into an array using a list of costs as defined in question 2a above.
4. Describe the IF statement and the different logical operators used in it. What alternative paths can be taken when a condition evaluates true or false?

5. Write a FOR—NEXT statement to generate
  - a) integer values from 1 to 25
  - b) even numbers from 2 to 100
  - c) a set of real numbers from .001 to .015 in increments of .001Print the results of each of the above loops.
6. Describe the benefits of a GOSUB over the GOTO statement.
7. Explain how random numbers may be generated. Write a statement to produce a random number between 1 and 10 inclusive. How would you get a random number between 5 and 15 inclusive?
8. What are some of the characteristics of a good game program?
9. Why are time delays sometimes necessary in a program? How are they created?
10. When designing a relatively complex program why is it best to first design the formats for input and output? What type of information will be needed to define input and output when only the screen and keyboard are used?
11. What is the purpose of English or pseudo code? How can it help in program development?



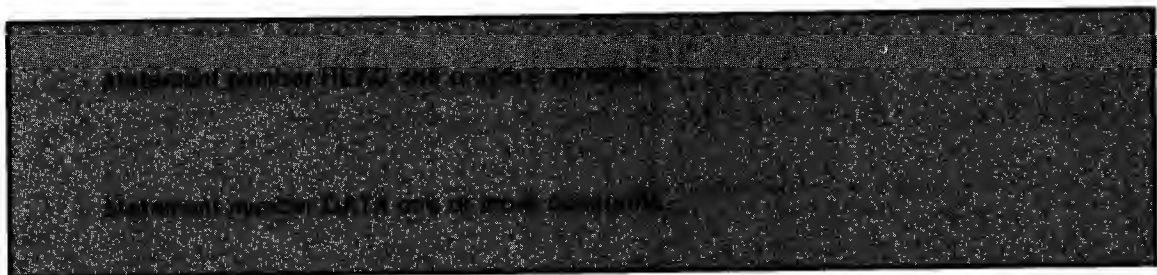


# 5

## ***More on Input and Output***

**M**any programs for business and personal use rely on the PET/CBM's input and output capability. We have already used the INPUT statement for the input of data from the keyboard and the PRINT for screen output. In this chapter we are going to look at the READ statement, which will let us read data that is directly recorded within the BASIC program. Secondly we will examine some more advanced features of the PRINT statement that will give us greater control over the formatting of screen output.

### ***READ—DATA***



The READ statement is used to access data stored in a DATA statement. The DATA statement permits the program to store frequently-used data within the program and access that data efficiently. A READ has a list of one or more variables separated by commas. Each time the READ is executed in the program, data is selected from the DATA statement in the order in which it appears. For example, the statements:

```
10 READ N,S,A$  
20 DATA 25,1.75,BOX
```

would cause the value 25 to be read from the DATA statement and assigned to the variable N. S receives the value 1.75 and A\$ is given the value BOX. Notice that string data may be given with the quotes ("BOX") or without (BOX).

When a DATA statement or statements contain more data than the READ accesses, the additional data will be read the next time the program reaches a READ.

## CREATING A BAR CHART

This program will use DATA statements to provide the values for bar charts to be displayed on the screen. The program reads a series of five numbers and produces a simple bar graph from them. Each number represents a quantitative measurement that directly affects the length of the bar representing that number.

```
10 FOR I=1 TO 5
20 READ N
30 FOR K=1 TO N
40 PRINT "[rvs][sp][rvs off];
50 NEXT K
60 PRINT: PRINT
70 NEXT I
80 DATA 13,5,26,8,34
```

loops N times

range of inner loop

range of outer loop

leaves space between each bar

This program uses nested FOR loops; one to control the reading of the data (outside loop) and one to print the graph (inside loop). The outside loop varies I from 1 to 5 to read each of the five numbers from the DATA statement one number at a time. As each number is read in 20 it is stored in N. N is then used in the K loop (statement 30) to control the printing of the bar.

The bar is printed in 40 by using the reverse space within a character string. This is typed as quote, reverse, space, reverse off, quote. The semicolon, at the end of the PRINT statement, causes the reverse characters to continue printing as a solid line without spaces. The number of times it prints is controlled by the K loop. The output from this program is shown in figure 5.1.

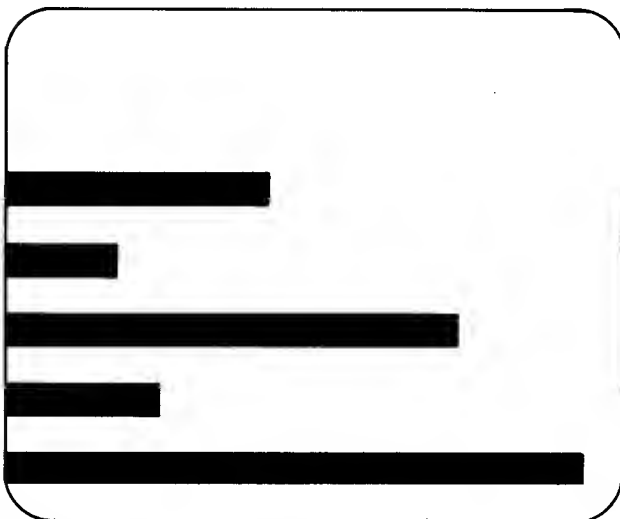
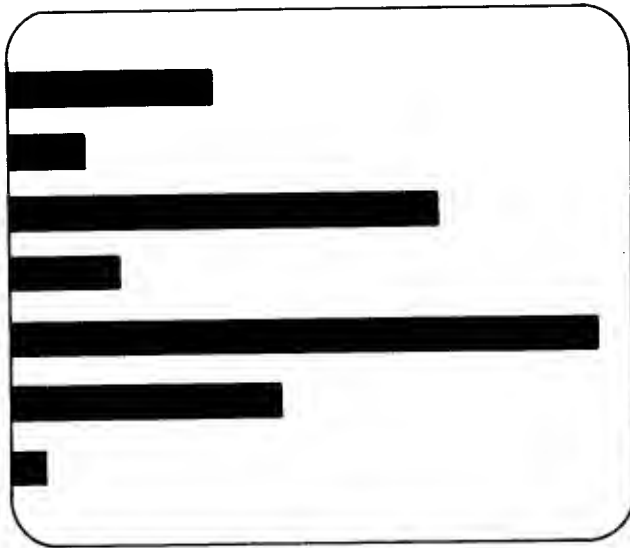


Figure 5.1 Simple bar chart output



**Figure 5.2** Bar chart output with variable data

One of the problems we sometimes face with the READ statement is not knowing when there is no more data to read. In the previous program we knew exactly how much data to read but this is not always the case.

Suppose the data sometimes consists of 5 numbers but other times 8 or 12 numbers. A simple solution is to place a dummy value at the end of the data to denote the end. The program can then test for this value with an IF statement. Ordinarily the dummy value should be something that cannot be confused with real data. Here is another approach to solving the bar graph with the output in figure 5.2.

```

10 READ N
20 IF N=99 THEN STOP
30 FOR K=1 TO N
40 PRINT "[rvs][sp][rvs off]";
50 NEXT K
60 PRINT: PRINT
70 GOTO 10
80 DATA 13,5,26,8,34,17,3,99

```

terminates program  
when last value is read

dummy  
value

### WEIGHTED AVERAGE WITH DATA

For one more example let's look at the weighted average program again and consider an easier way to assign the maximum marks and weights to the arrays M and W by using DATA statements. The following code reads these values from data statements and assigns them to the array.

```

10 DIM M(5),W(5)
20 FOR I=1 TO 5
30 READ M(I),W(I)
40 NEXT I
50 DATA 30,2,25,1,50,2,10,3,75,2

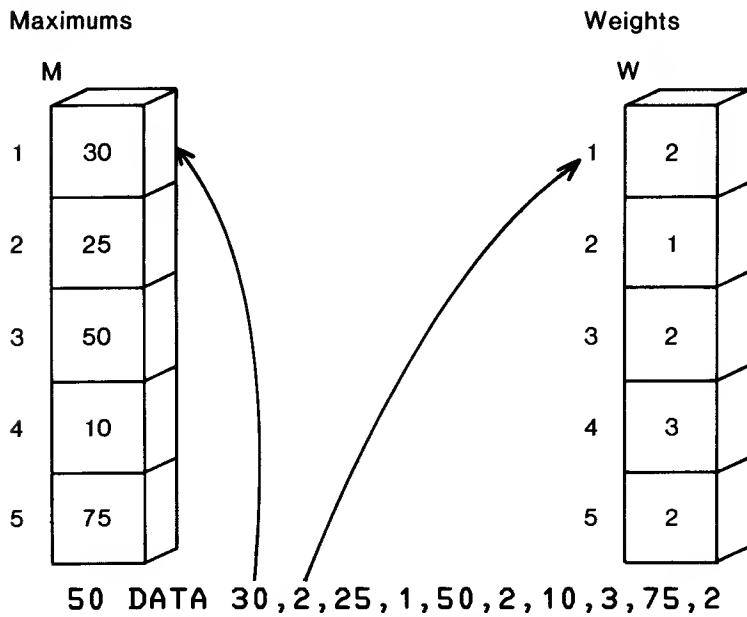
```

read a maximum  
and a weight

maximum

weight

The DATA statements contain a maximum mark followed by its weight. Thus the READ references M(I) and then W(I) to read each of these values, in order, from the DATA statement into the arrays, as follows:



## RESTORE



As data is read from DATA statements the computer keeps track of each value in memory by using an internal data pointer. Although this pointer is not visible to the BASIC programmer the BASIC interpreter in the ROM makes continuous reference to it.

The RESTORE statement can be used in BASIC to set the data pointer back to the first data item in the DATA statements. This feature can be useful if you need to read through the same set of data more than once within the program.

It's not unusual in a Computer Assisted Instruction (CAI) program to give a series of instructions to a student and then ask questions about that same information. In the following program we will do a simplified version of CAI by teaching the names of the five oceans on Earth and then asking for the student to respond by typing in the names. Beware! This is not intended to be an example of good CAI. Far from it; but it does show how to use the RESTORE statement, which is our current objective.

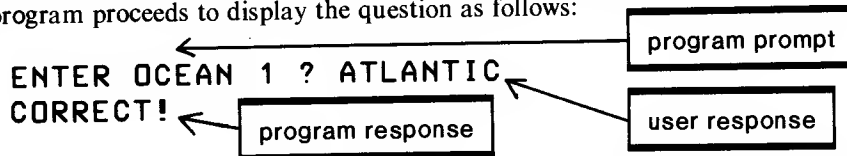
A DATA statement will be used to supply the names of the oceans. These names are first displayed as follows:

THERE ARE FIVE OCEANS ON PLANET EARTH.  
THEY ARE CALLED:

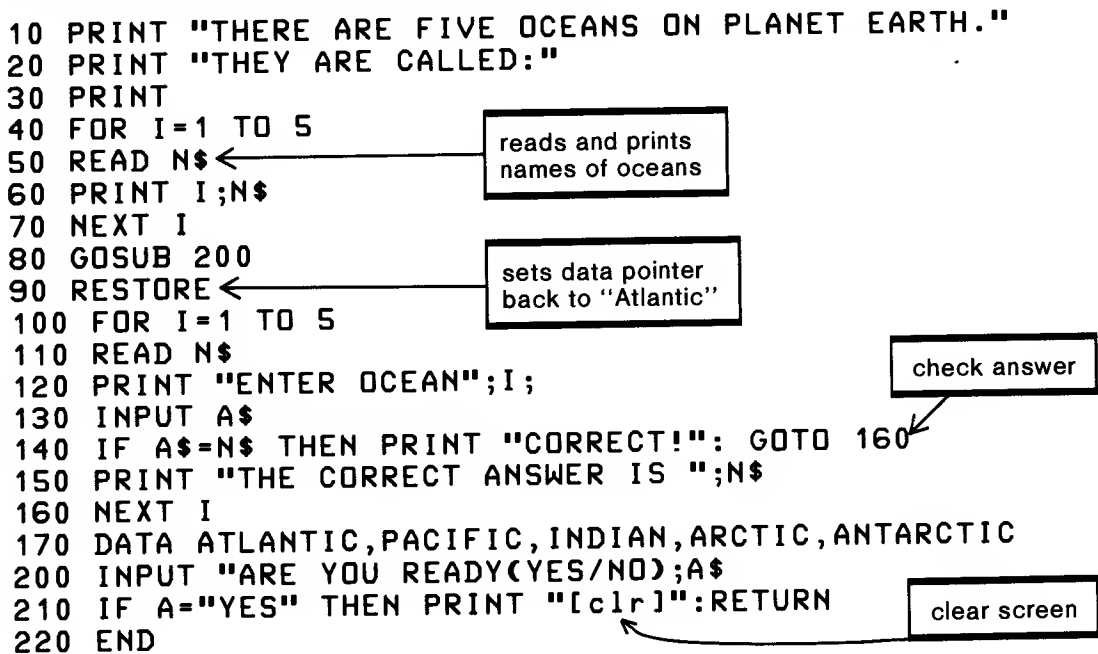
- 1 ATLANTIC
- 2 PACIFIC
- 3 INDIAN
- 4 ARCTIC
- 5 ANTARCTIC

ARE YOU READY(YES/NO) ?

After the names have been displayed the program RESTOREs the data pointer so that the data may be read again. When the user has had time to review these names and types YES to the prompt, the program proceeds to display the question as follows:



If the user had entered an incorrect answer the program would then display the correct answer and proceed to the next ocean. Here is the program:



## MORE ABOUT PRINT

In chapter 3 we looked at the basic features of the PRINT statement and found the capability to solve a variety of problems relating to screen output. Now we are going to add several additional features to the PRINT that will give us more flexibility in our programming.

### Cursor Controls











The cursor controls discussed in chapter 3 may also be used as commands within a string in the PRINT statement. These controls are the CLR, HOME, Cursor Down, Cursor Up, Cursor Right, and Cursor Left keys. These characters must be entered as part of a character string by first pressing the quote (") and then keying the appropriate cursor character. For example, to clear the screen and position the print to the fourth line, type the statement:

```
100 PRINT "[clr dn dn dn]"
```

The square brackets in this statement indicate the enclosed characters are cursor controls and are entered by pressing the CLR key once, followed by the Cursor Down Key three times. The square brackets are not typed, nor are the spaces between clr and dn.

This is how statement 100 would look on your screen:



Keyboard	Graphic on Screen
CLR	 Reverse Heart
HOME	 Reverse S
 Cursor Down	 Reverse Q
 Cursor Up	 Reverse Solid Circle
 Cursor Right	 Reverse Right Square Bracket
 Cursor Left	 Reverse Vertical Line

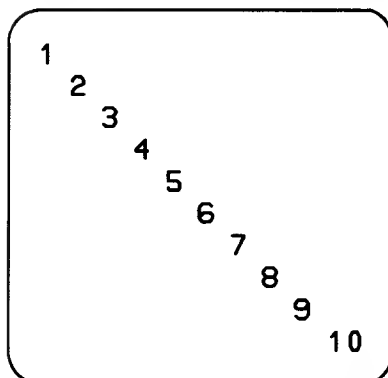
**Figure 5.3** Cursor controls in a string

Figure 5.3 shows the cursor controls and the equivalent graphic displayed when they are entered into a string.

To get a feel for the use of cursor controls in the PRINT statement try the following examples:

```
100 FOR I=1 TO 10
110 PRINT "[dn rt]";I;
120 NEXT I
```

These statements should give the output:



across and down the screen. Try some other variations of the cursor controls and see what results you can achieve.

Now try the code:

```
100 FOR I=1 TO 1000
110 PRINT "[clr dn rt]";I;
120 NEXT I
```

When you run this code you'll get the numbers from 1 to 1000 displaying rapidly like a counter at the top left corner of the screen. The clear command in the string causes the screen to be cleared after each print operation and the cursor controls bring it back to the previous position to display the next number. Thus each new value quickly is cleared from the screen and is replaced by the next, giving a counter effect.

## **TAB**

TAB is a function of BASIC that is used with the PRINT statement to move the cursor to a specified location on the line. The format used is

TAB(n)

where n may be any value from 0 to 255. When TAB is used the cursor moves to position  $n + 1$  of the line. For example, TAB(12) would move the cursor to position 13.

The TAB function is always used in a PRINT statement prior to the value that is to be printed. TAB is followed by a semicolon and then the value to be printed at the location specified.

```
10 PRINT "12345678901234567890"
20 PRINT "ABCD";TAB(8);"B"
```

The above use of TAB will create the following output on the screen.

```
12345678901234567890
ABCD      B
```

TAB is useful when you want to display outputs at a specific position on the screen, thus leaving a precise amount of space between each field. Multiple TABs may be used in one PRINT statement as follows:

```
20 PRINT "ABCD";TAB(8);"B";TAB(20);N;TAB(26);A$
```

Assuming a value of 28 for N and "XYZ" for A\$ the following output would be produced.

Columns				
0	1	2	3	
1234567890	1234567890	1234567890		
<hr/>				
ABCD	B	28	XYZ	

These values seem to follow the pattern discussed above except for N's value of 28. We would expect it to be in column 21 but instead it is in column 22. The reason is quite simple. All numeric values reserve a position for the sign. In this case the sign is plus and so it does not display but rather leaves column 21 blank. If the value had been  $-28$  then column 21 would be occupied with the minus sign as expected.

## **SPC**

The SPC (space) function is similar to TAB except that SPC defines the number of spaces to leave between the previous output on the line and the one to follow.

```
10 PRINT "12345678901234567890"
20 PRINT "ABCD";SPC(5);"B"
```

The above code gives the results:

```
12345678901234567890
ABCD      B
```

A common problem with both TAB and SPC is the space that replaces the sign for positive numeric values. Thus the statement

```
10 PRINT "SUM";SPC(3);S
```



will actually leave four spaces; three for the SPC command and one for the sign if S is a positive value. If S is negative there will be three spaces and a minus sign before the value.

## METRIC CONVERSION PROGRAM

The next program we want to tackle is one to convert from English to Metric measure or the reverse. Figure 5.4 shows the conversion factors needed to do the calculations for the appropriate measurement. Values to convert from English to Metric are given. Conversion in the opposite direction is achieved by using the inverse of these values.

By observation we can determine that this table consists of three parts: Linear, Square, and Liquid and Dry Measure. These parts form a basis for the logical organization of our program. Initially the program will read the table information from DATA statements and load it onto arrays for use by the program.

So far in this book we have used English code for the purpose of developing program logic. Actually, the use of English or Pseudo code for developing program logic is quite new. Prior to its use a popular program development tool was the flowchart.

One of the advantages of flowcharting is the visual or graphic aspect, which sometimes helps you to visualize a solution. Since some people are more verbally oriented and others image-oriented the flowchart might seem to be a good alternative if you are the second type of person. One reason for the move away from flowcharts has been the stress on a technique called structured programming. That, however, is a subject for another book.

Figure 5.5 shows the various symbols used for flowcharting.

Figure 5.6 shows the flowchart for the logical solution of this problem.

English	Metric	Factor
Inches	Centimeters	2.54
Feet	Meters	.3048
Yards	Meters	.9144
Miles	Kilometers	1.6093
Square Inches	Square Centimeters	6.452
Square Feet	Square Meters	.0929
Square Yards	Square Meters	.8361
Square Miles	Square Hectometers	259
Dry Quarts	Liters	1.101
Liquid Quarts	Liters	.9463
Liquid Gallons	Decaliters	.3785
Pecks	Liters	8.81
Bushels	Hectoliters	.3524

Figure 5.4 Metric conversion factors

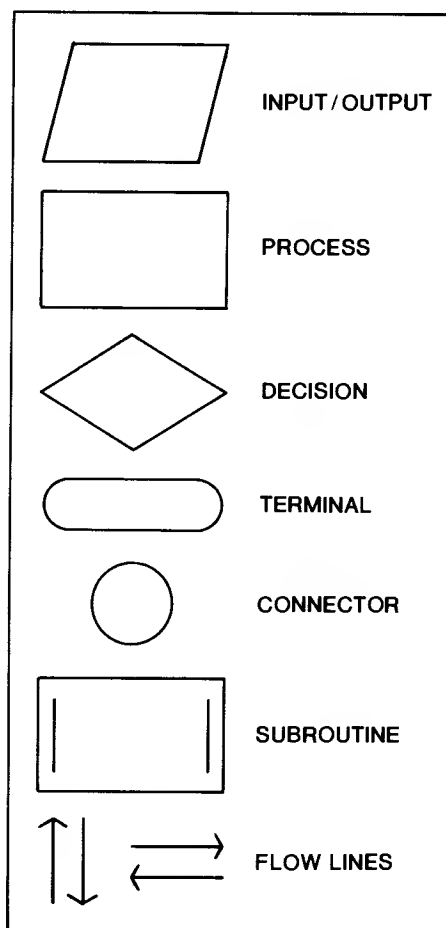


Figure 5.5 Flowchart symbols

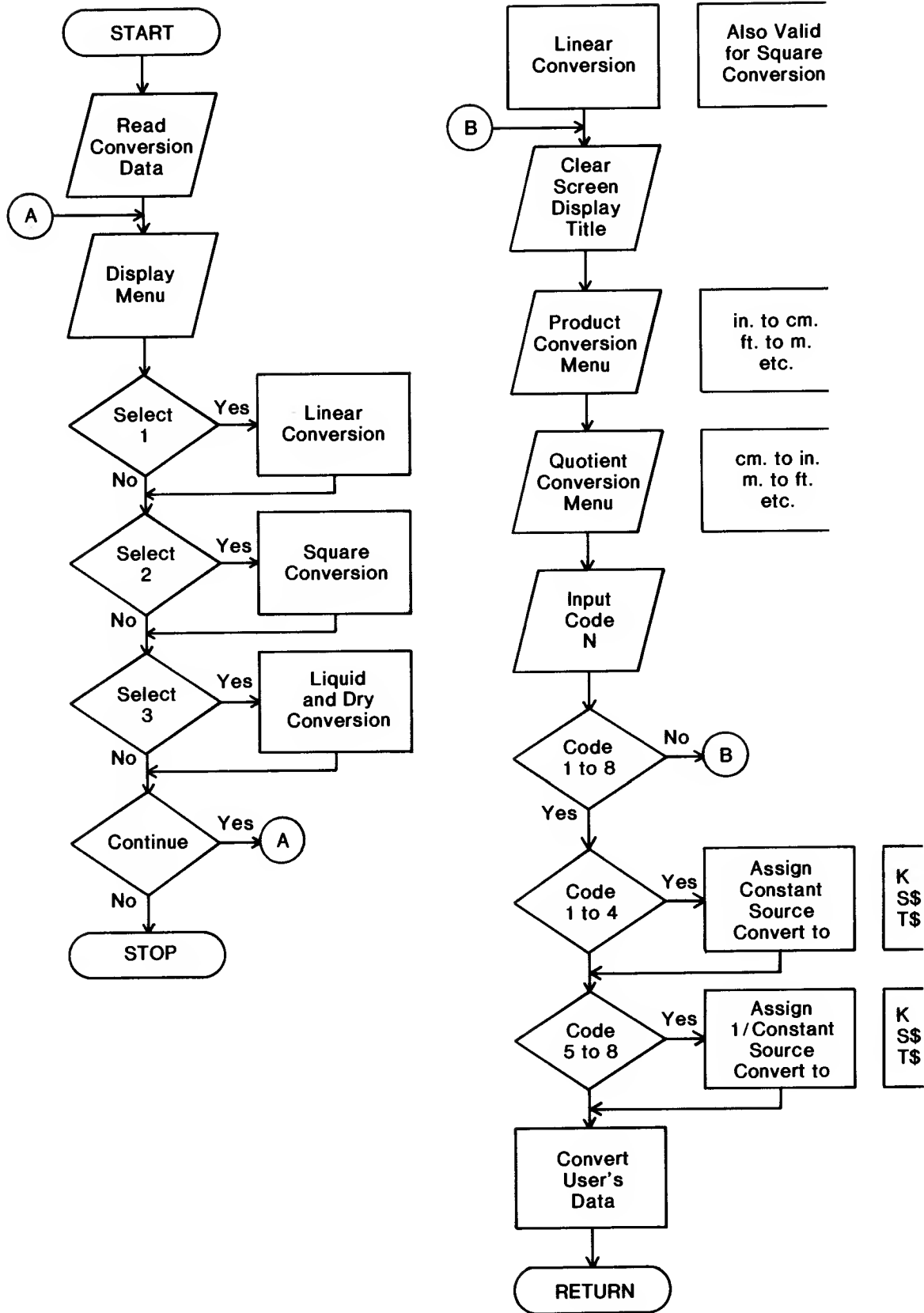


Figure 5.6 Metric conversion flowchart

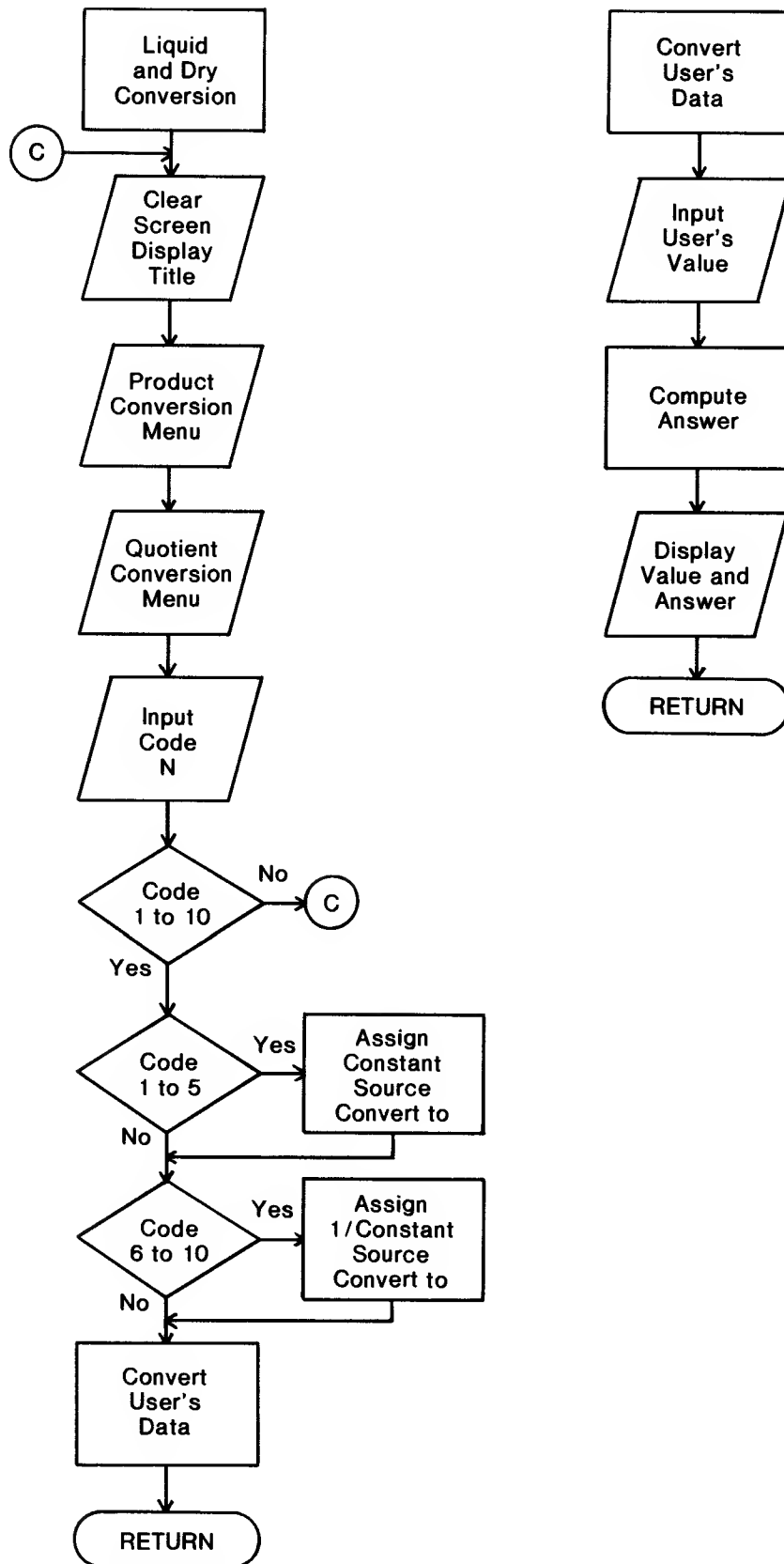


Figure 5.6 (continued)

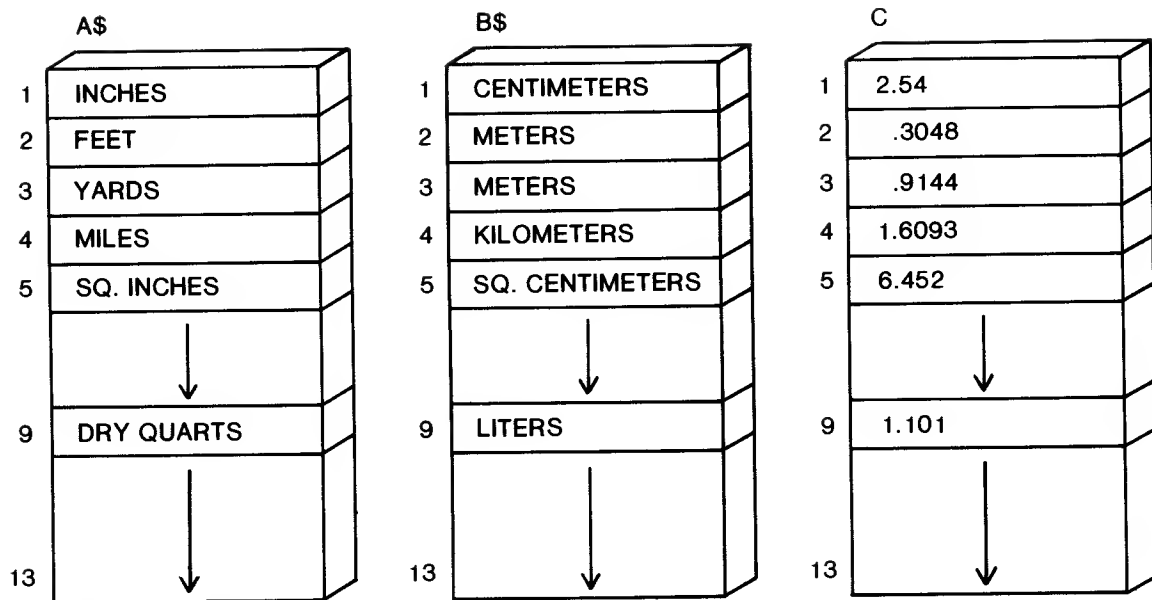


Figure 5.7 Arrays for conversion data

This program is a natural for arrays and, of course, DATA statements are used heavily. The arrays are organized as shown in figure 5.7. Notice at this time the program makes no distinction between the different categories of measure. Array A\$ contains the description of English measurements, B\$ the Metric descriptions, and C contains each conversion factor for the related English and Metric conversion.

These arrays are first used to produce the menu for selecting the required conversion. For the Linear Conversion Menu the statements

```
360 FOR I=1 TO 4
370 PRINT TAB(4);I;"-";A$(I);" TO ";B$(I)
380 PRINT
390 NEXT I
```

are used to print the sequence:

- 1 - INCHES TO CENTIMETERS
- 2 - FEET TO METERS
- 3 - YARDS TO METERS
- 4 - MILES TO KILOMETERS

The variable I has a two-fold purpose here. One, it is used to print the digits 1 to 4 in the menu and two, it selects the position of arrays A\$ and B\$ for printing the English and Metric descriptions.

The second half of the Linear menu permits conversions from Metric to English and uses the code

```
400 FOR I=1 TO 4
410 PRINT TAB(4);I+4;" - ";B$(I);" TO ";A$(I)
420 PRINT
430 NEXT I
```

These statements print menu items 5 to 8 as follows:

- 5 - CENTIMETERS TO INCHES
- 6 - METERS TO FEET
- 7 - METERS TO YARDS
- 8 - KILOMETERS TO MILES

Again I is used in two ways. This time the expression  $I + 4$  is used in the Print statement to print digits 5 to 8 so that when I is 1 the number 5 will print, 2 causes 6 to print, and so on. Subscripts in this case are still 1 to 4 but the array references are now reversed in the print, giving B\$ and then A\$.

This basic approach is used for each menu, except that the starting and ending values of the FOR loop are adjusted to correspond to the location of the names in the arrays. For Square Conversion the values are 5 to 8 as indicated in the following code.

```

550 FOR I=5 TO 8
560 PRINT TAB(4);I-4;"-";A$(I);" TO ";B$(I)
570 PRINT
580 NEXT I
590 FOR I=5 TO 8
600 PRINT TAB(4);I;" - ";$(I);" TO ";A$(I)
610 PRINT
620 NEXT I

```

Liquid and Dry Conversion uses the values 9 to 13, thus relating to the positions of these quantities in the arrays.

Once the appropriate menu has been displayed the user of the program enters one of the values to select the conversion required. This number is stored in N. From the value of N three things are accomplished. The variable K is assigned the conversion constant from the array C; S\$ is assigned the description of the source of conversion from either array A\$ or B\$; and T\$ is assigned the description we are converting to, from either A\$ or B\$.

For example, if Linear Conversion is selected and N was given 6, indicating meters to feet, K will be assigned  $1/C(N-4)$ , which reduces to  $1/C(2)$  thus referring to the value .3048 in position 2 of array C. S\$ will be given the value at position 2 of B\$(METERS), and T\$ is assigned position 2 of A\$(FEET). The following code does this operation.

```

470 IF N<=4 THEN K=C(N):S$=A$(N):T$=B$(N)
480 IF N>4 THEN K=1/C(N-4):S$=B$(N-4):T$=A$(N-4)

```

In the second menu, Square Conversion, the values of N may also be 1 to 8 but the positions of these conversions in the arrays are from 5 to 8. In this case menu responses of 1 to 4 correspond to positions 5 to 8. By simply increasing N by 4 the program points to the right place. Responses 5 to 8 luckily also point to positions 5 to 8 so no adjustment is needed. Here is the code.

```

660 IF N<=4 THEN K=C(N+4):S$=A$(N+4):T$=B$(N+4)
670 IF N>4 THEN K=1/C(N):S$=B$(N):T$=A$(N)

```

Finally Liquid and Dry Conversion can have values of 1 to 10 for N. The first five relate to positions 9 to 13 of the arrays. This correspondence is achieved using  $N+8$  for the subscript. Menu responses 6 to 10 use a subscript of  $N+3$  to reference array positions 9 to 13.

```

850 IF N<=5 THEN K=C(N+8):S$=A$(N+8):T$=B$(N+8)
860 IF N>5 THEN K=1/C(N+3):S$=B$(N+3):T$=A$(N+3)

```

The complete metric conversion program is shown in figure 5.8.

### ***SIMPLE PAYROLL PROGRAM***

Payroll is a traditional data processing activity that can be implemented successfully on a microcomputer. Although no attempt is made here to write a complete payroll program some of the principles are demonstrated even in this simple version.

One of the needs of a payroll system is to withhold income tax for each pay period. This is done by using a table that supplies the amount of tax to be deducted for different levels of income. A table

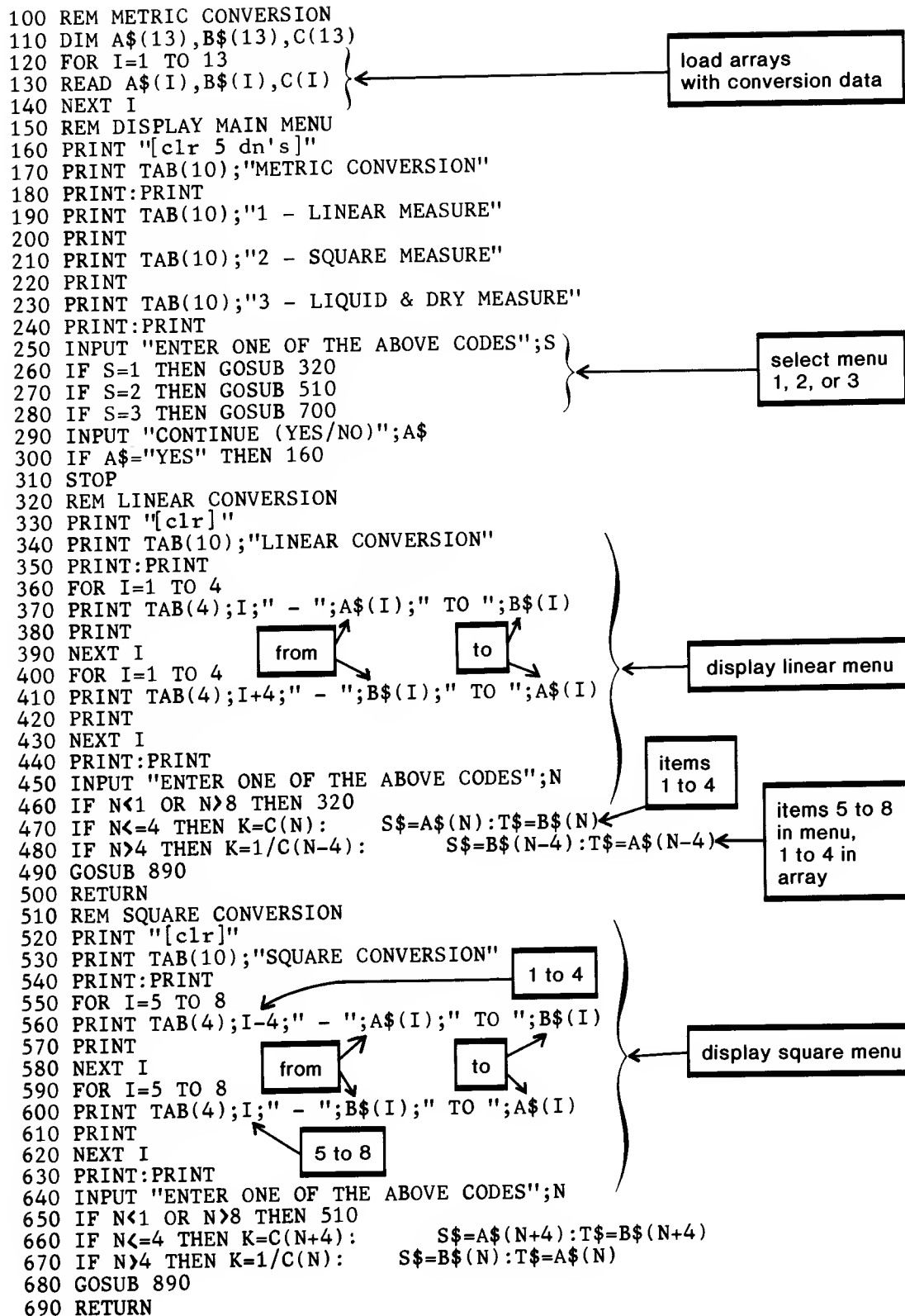


Figure 5.8 Metric conversion program

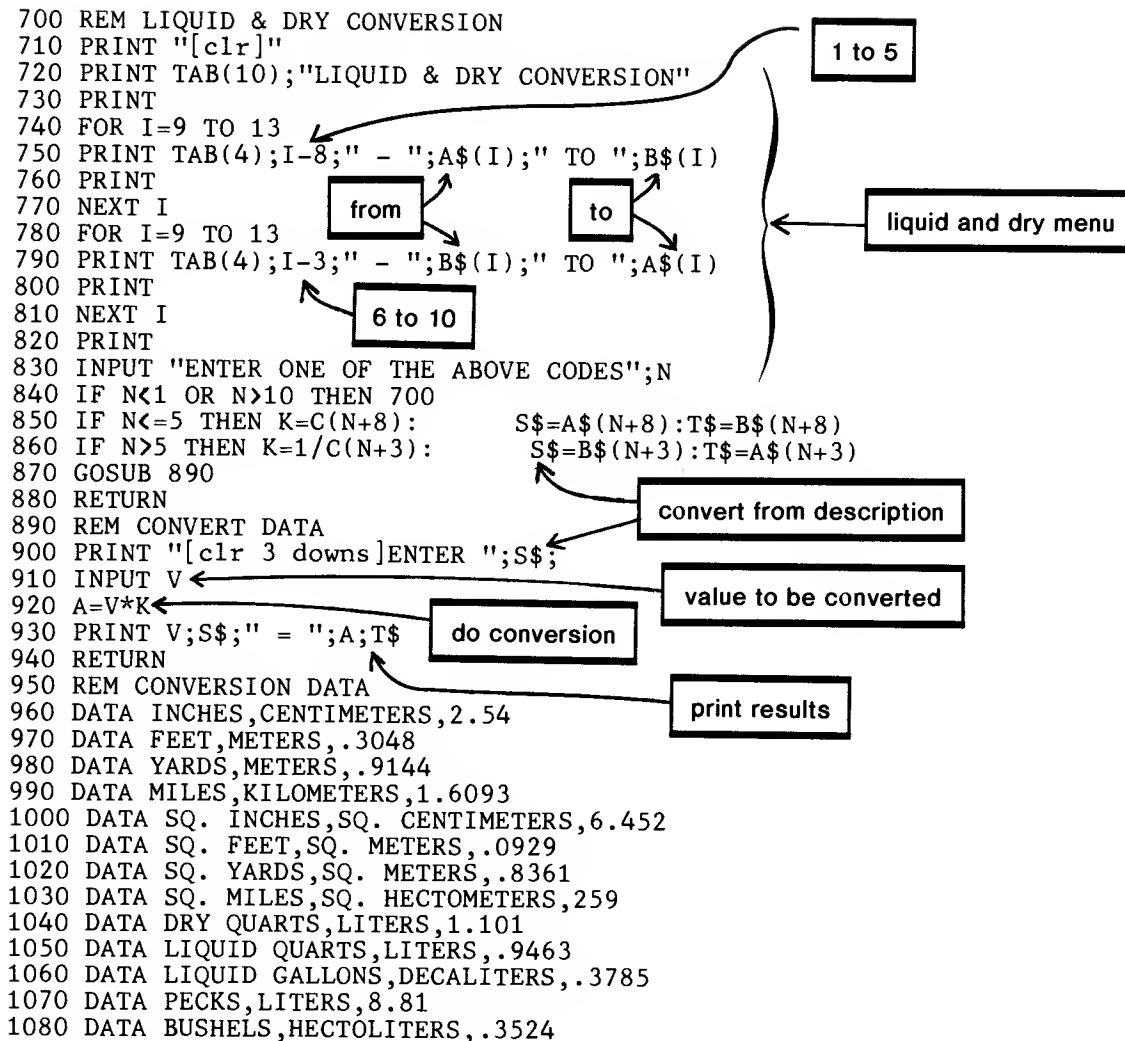


Figure 5.8 Metric conversion program (continued)

Gross Amount	Withholding Amount	Percent
0- 46	0	0
46-127	0	15
127-210	12.15	18
210-288	27.09	21
288-369	43.47	24
369-454	62.91	28
454-556	86.71	32
556-999	119.35	37

Figure 5.9 Weekly withholding tax table

of this type for weekly deductions is shown in figure 5.9. The first column represents the maximum gross salary. Column two is the amount of tax to be withheld and column three gives a percent to be applied to an amount in excess of the minimum for that category.

The table is used by finding the category for a given gross salary and selecting the tax to be deducted. For instance, a gross of \$300.00 would require a tax of 43.47 plus 24% of the remaining \$12.00 (derived from 300.00 - 288.00).

A flowchart for the solution is in figure 5.10, the program is in figure 5.11, and a sample output in figure 5.12. The program first loads the tax table into arrays GA (Gross Amount), WA (Withholding Amount), and PC (Percentage). Notice that the data and array GA contain only the high value for

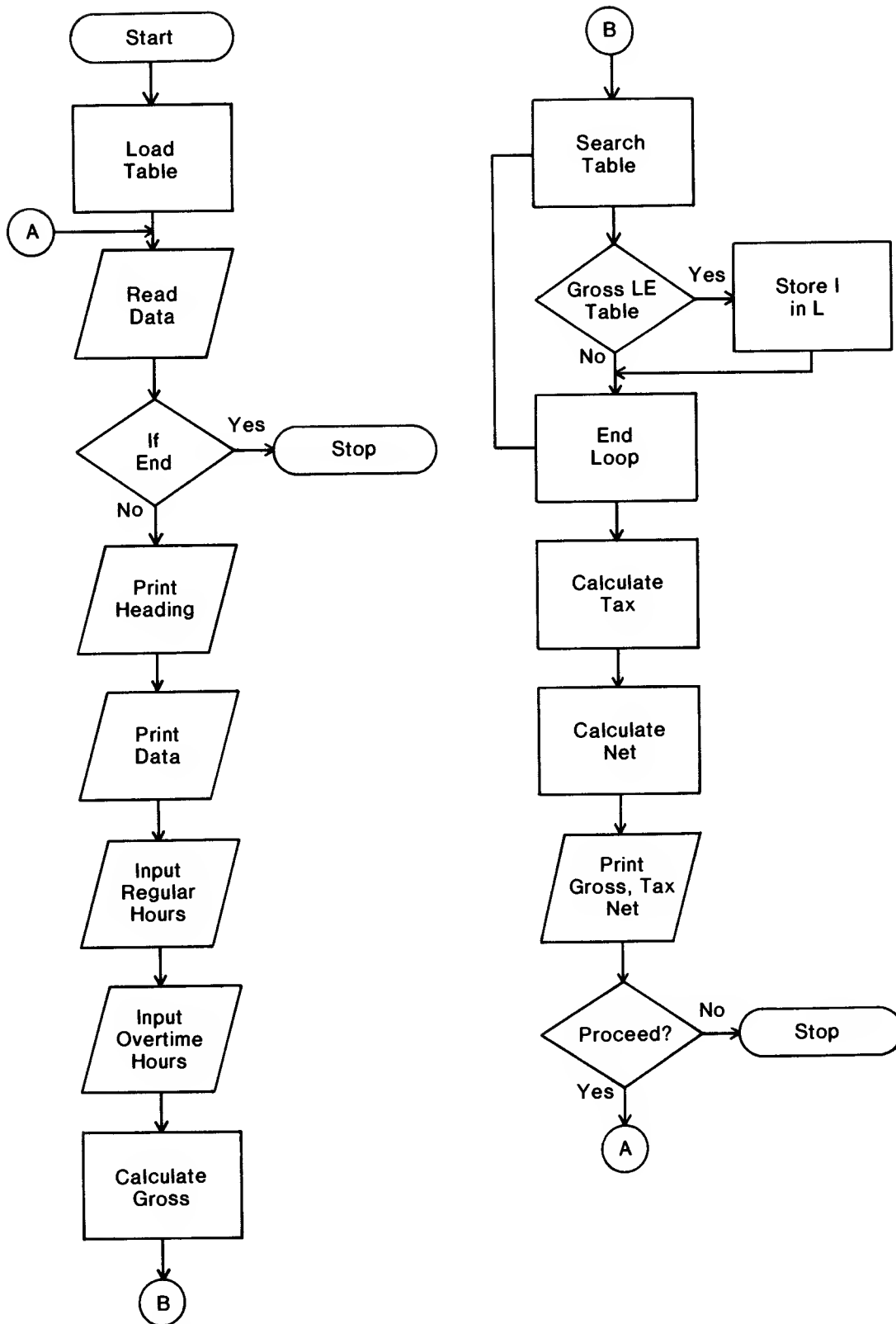


Figure 5.10 Flowchart for simple payroll



```

100 REM SIMPLE PAYROLL
110 DIM GA(8),WA(8),PC(8)
120 REM LOAD TAX TABLES
130 FOR I=0 TO 8
140 READ GA(I),WA(I),PC(I)
150 NEXT I
160 REM TAX TABLE DATA
170 DATA 0,0,0
180 DATA 46,0,0,127,0,.15,210,12.15,.18
190 DATA 288,27.09,.21,369,43.47,.24
200 DATA 454,62.91,.28,556,86.71,.32
210 DATA 999,119.35,.37
220 REM PROCESS PAYROLL
230 PRINT "[clr]"
240 READ E,D,N$,R
250 IF E=999 THEN STOP
260 PRINT "EMPLOYEE   DEPT   NAME           RATE"
270 PRINT
280 PRINT E,D;N$,R
290 PRINT
300 INPUT "ENTER REGULAR HOURS WORKED";HW
310 INPUT "ENTER OVERTIME HOURS";HO
320 PRINT
330 GS=HW*R+HO*1.5*R
340 REM FIND TAX RATE LOCATION IN TABLE
350 FOR I=1 TO 8
360 IF GS<=GA(I) THEN L=I:I=8
370 NEXT I
380 TAX=WA(L)+((GS-GA(L-1))*PC(L))
390 NS=GS-TAX
400 PRINT "GROSS SALARY";GS
410 PRINT
420 PRINT "TAX DEDUCTED";TAX
430 PRINT
440 PRINT "NET SALARY";NS
450 PRINT:PRINT
460 INPUT "PROCEED (YES/NO)";X$
470 IF X$="YES" THEN 230
480 STOP
490 DATA 123,100,JOSEPH SMITH,8.25
500 DATA 124,100,JANE SIMON,9.15
510 DATA 128,110,CARLO ROUSE,8.55
520 DATA 130,110,PETER SHANE,7.90
530 DATA 999,9,END,9

```

Figure 5.11 Simple payroll program

each category. This simplifies the array and yet the lower value is always available in the preceding array element. Since a problem could occur with the first entry a value of zero is stored in element zero of each array (Remember arrays begin with element zero in BASIC).

Next each employee's record is read and displayed on the screen and the user is requested to enter the regular and overtime hours. The program then calculates Gross Salary (GS) in 330 and looks for the appropriate position in the table in statements 350 to 370. The location is stored in L. Now line 380 calculates the tax. Finally the Gross, Tax, and Net Amounts are displayed.

EMPLOYEE	DEPT	NAME	RATE
123	100	JOSEPH SMITH	8.25

ENTER REGULAR HOURS WORKED? 38  
ENTER OVERTIME HOURS? 5

GROSS SALARY 375.375

TAX DEDUCTED 64.695

NET SALARY 310.68

PROCEED (YES/NO)?

**Figure 5.12** Sample payroll screen output

### **REVIEW QUESTIONS—CHAPTER 5**

1. Explain the use of READ and DATA statements. Why would they be used instead of the INPUT statement?
2. If a program needs to read the same data more than once during processing, give two ways this may be done in BASIC.
3. Why would we want to use cursor control characters in string data or in a PRINT statement?
4. Give an example not used in the chapter of cursor controls used in a string.
5. Explain the use of the TAB and SPC functions.
6. Why are flowcharts sometimes used by programmers when developing program logic?
7. What are the advantages and disadvantages of flowcharts compared to English code? Which of these methods do you prefer?
8. Consider the program for metric conversion in this chapter. What other conversions might be useful in a program of this type? How would you revise the program to include additional conversion categories?



# 6

## **Advanced BASIC**

**M**any of the statements in this chapter could be implemented using an appropriate combination of BASIC statements previously discussed. Why then do we bother to use these new statements? The major reason is that in certain circumstances these new commands are easier to use. In other situations the new instructions are faster to execute and more efficient than the old way. So let's get on with the new bag of tricks!

### **GET**



The GET statement is used to read a single character from the keyboard without the need to press Return after the character has been entered (figure 6.1). The GET in a program looks like

```
100 GET A$
```

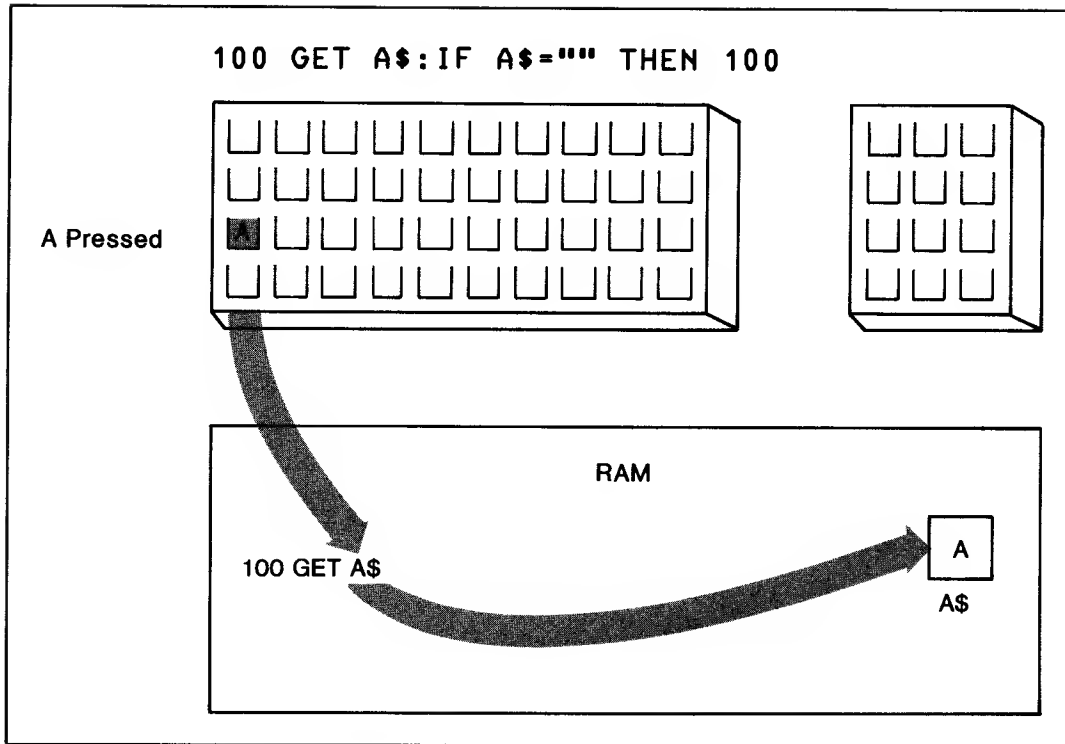
But, since the action is instantaneous a delay loop is required to wait for a user response and to check the variable for a value. This test is done by comparing A\$ to a null string. If A\$ is still null (or empty) a branch back to the GET is taken.

Whenever a key is pressed the value typed enters A\$, which is no longer null. At this time the program will continue at the statement following 100.

A common use for GET is in response to a question requiring YES or NO as an answer. Try the following example:

```
100 PRINT "DO YOU WANT INSTRUCTIONS (Y/N)?":  
100 GET Z$:IF Z$="" THEN 110  
120 IF Z$="Y" THEN GOSUB 500
```

First the program prints a message indicating the type of response expected, either a Y representing Yes or an N for No. Then the GET statement is placed in a waiting loop until the user presses a key.



**Figure 6.1** GET statement

The value of the key pressed goes into Z\$ and statement 120 then tests Z\$ for a Y response. If the user of our program types the entire word YES only the first letter will be accepted and the remainder of the word is ignored.

### ON—GOSUB

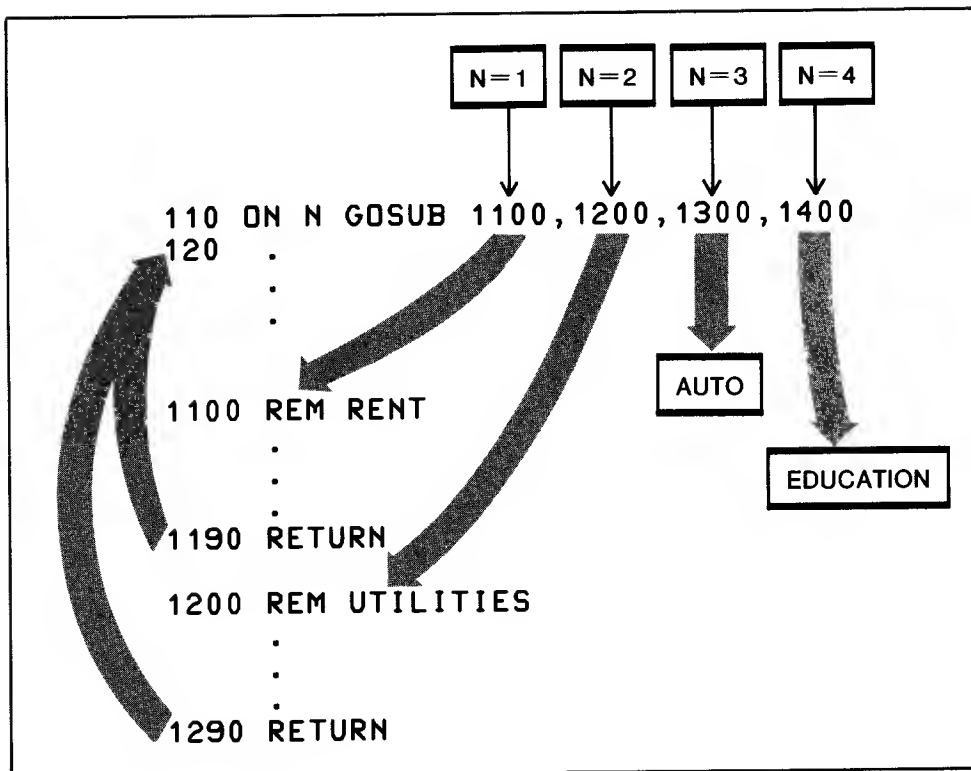


The ON—GOSUB is a single statement that replaces a series of IF—THEN—GOSUBs when the values to be tested are numeric and consecutive. Given a menu

- 1 - RENT
- 2 - UTILITIES
- 3 - AUTO
- 4 - EDUCATION

ENTER ONE OF THE ABOVE CODES?

we could test the input response with four IF statements that each branch to the appropriate subroutine for processing. Assuming the program accepted the code with the command



**Figure 6.2** Using ON—GOSUB

```
100 INPUT N
```

the ON statement will process the response with only one statement.

```
110 ON N GOSUB 1100, 1200, 1300, 1400
```

The ON statement examines the variable (N) presented to it and depending on its value branches to one of the line numbers specified (figure 6.2). If N is 1, branching goes to 1100, the first line number. For a value of 2 the program branches to 1200 and so on. Any numeric variable name may be used instead of N and any number of line numbers specified up to a maximum of 255.

If the variable contained a value of 0 or if it exceeded the number of entries, for example a value of 5, then the ON continues at the next statement without branching.

Since we used the GOSUB the program will branch to the appropriate subroutine, such as 1300 for a value of 3 in N. When a RETURN is reached in subroutine 1300 control returns to the ON statement and passes to the next statement following line number 110.

### ON—GOTO



The only difference between ON—GOTO and ON—GOSUB is that ON—GOTO does not return from the location to which it has branched. Otherwise the statements function identically.

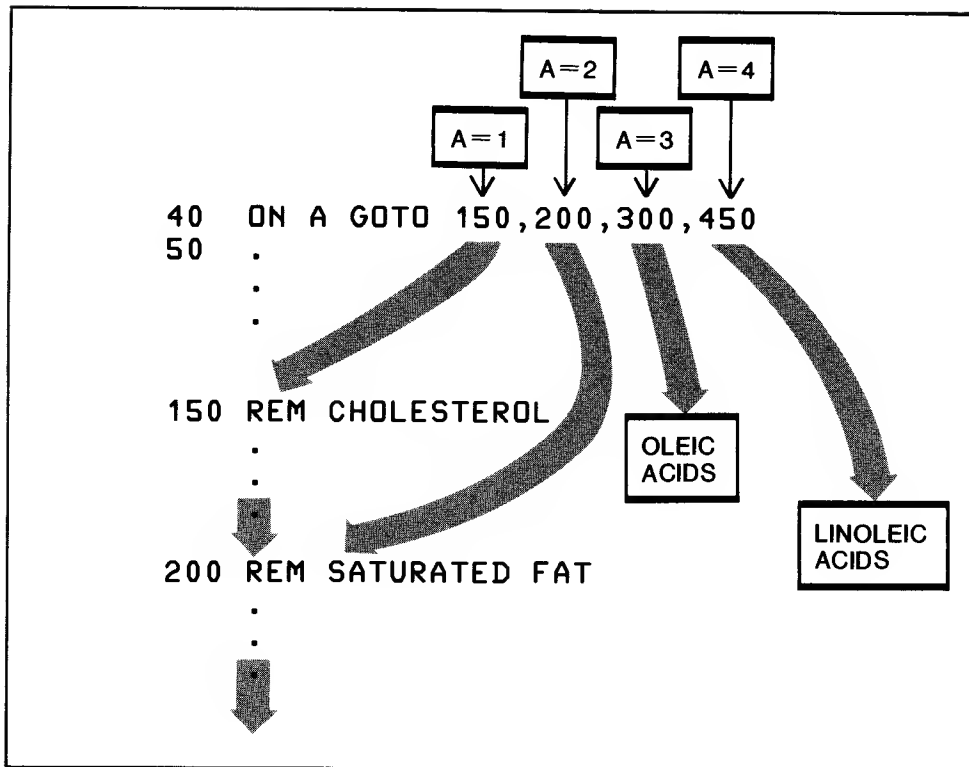


Figure 6.3 Using ON—GOTO

#### SELECT ONE CATEGORY FOR REDUCTION

- 1 - CHOLESTEROL
- 2 - SATURATED FAT
- 3 - OLEIC ACIDS
- 4 - LINOLEIC ACIDS

The response to this menu might be as follows:

```
40 ON A GOTO 150,200,300,450
```

If the response to the menu was 1 to select CHOLESTEROL the program would branch to statement 150, SATURATED FAT to 200, OLEIC ACIDS to 300, and LINOLEIC ACIDS to statement 450 as shown in figure 6.3.

#### POKE

statement number	POKE address	constant variable expression

The POKE statement places a value directly into a specified memory location. For example the statement

```
10 POKE 32768,83
```

would place a graphic heart symbol on the upper left corner of the screen. The value 32768 refers to the memory address and may take on values from 0 to 65535 inclusive, while the number 83 is the value being stored at this address. Eighty-three is PET/CBM machine code for the heart symbol (see Appendix E for PET/CBM Codes).

One use of POKE is to set the PET into certain modes such as upper- and lowercase letters. Try using

```
POKE 59468,14
```

Now, when you type on the keyboard, upper- and lowercase letters will display instead of just uppercase with graphics. On the more recent PET models typing a letter will result in lowercase, while a shifted letter will be uppercase, just like a typewriter. Older models may be the reverse of this depending on the level of ROM in the machine. To restore your PET back to graphics mode, type

```
POKE 59468,12
```

When you press Return, any lowercase letters on the screen will revert immediately to graphics.

The CBM is just the reverse of the PET. Normally the CBM is in letter mode but by using POKE 59468,12 it can be placed in graphics mode. Using POKE 59468,14 will get the CBM back to character mode.

The PET screen contains 1000 positions (figure 6.4) for displaying characters. These positions are addressed from 32768 to 33767. (The CBM has 2000 positions from 32767 to 34767). You can POKE any of these positions, as in figure 6.5, to create graphics or special effects. Try entering the following code for a simple example of this.

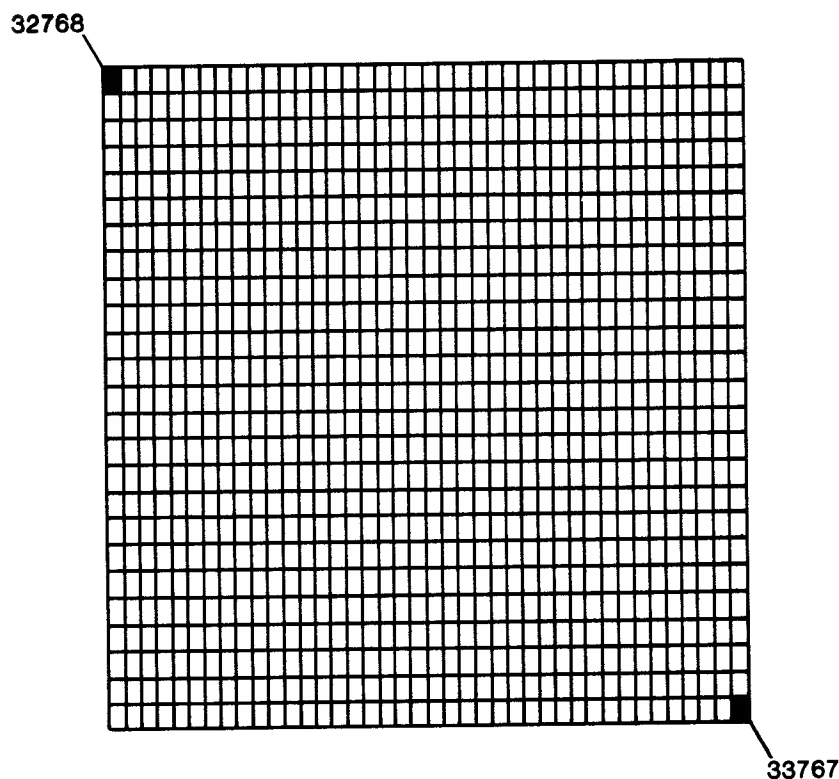


Figure 6.4 PET display screen addresses



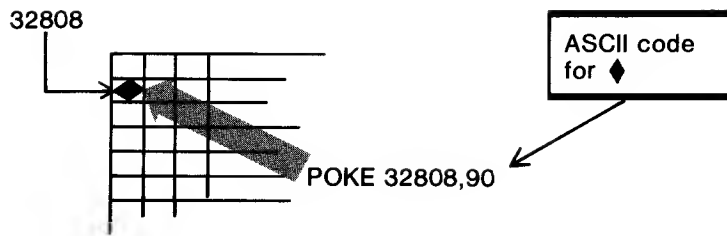
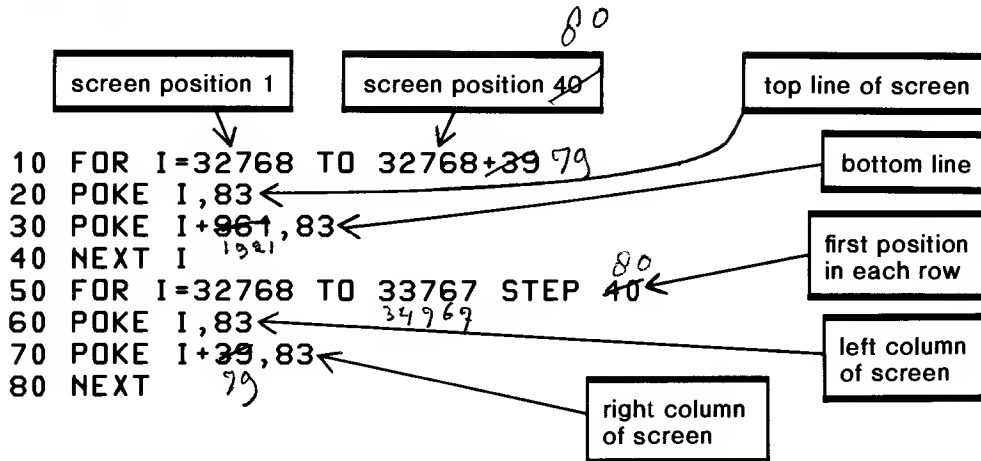
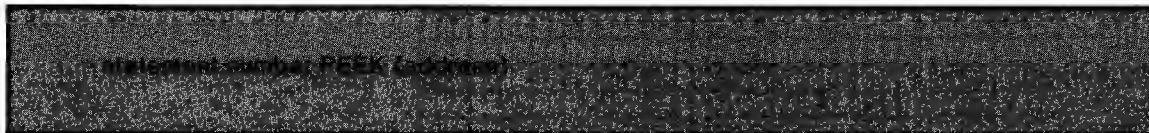


Figure 6.5 Poking screen memory



Now clear the screen and RUN this program. You should get a border of hearts around the outside edge of the screen. Although this could be done with the Print statement the POKE is faster than printing and permits very precise control of each screen position, which is particularly useful for animation. We will look at the use of POKEs in much greater detail in the chapter on Graphics and Animation.

## PEEK



The PEEK returns the contents of an address in memory. If the results from the previous program were left on the screen so that the border of hearts was still there, the command

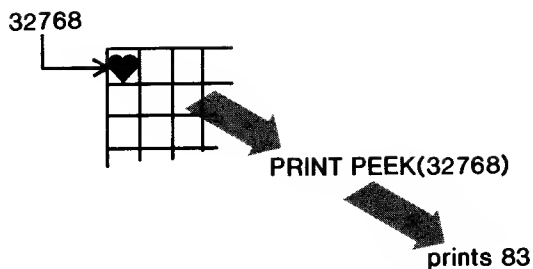
```
PRINT PEEK(32768)
```

would display the value 83 (figure 6.6), which is the ASCII code of the heart at memory address 32768. Of course if the heart was no longer there some other value would be displayed. The memory address could also be a variable as in

```
PRINT PEEK(I)
```

PEEK may return a value to a variable such as

```
120 M=PEEK(I)
```

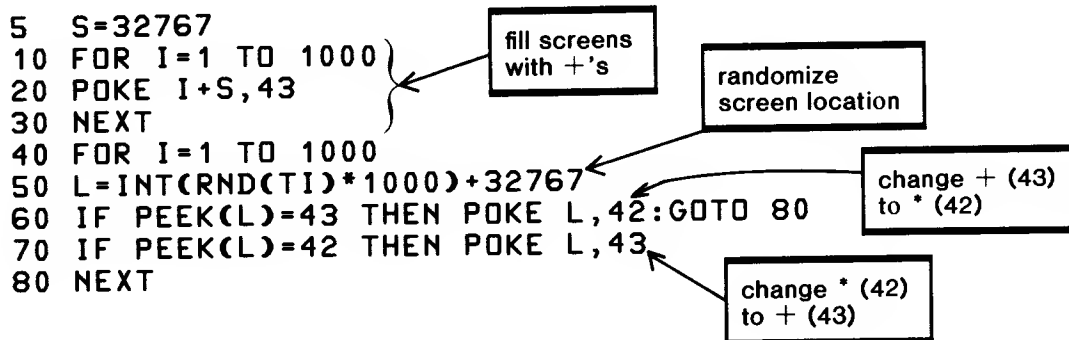


**Figure 6.6** Peeking at screen memory

where the result is assigned to a variable such as M. PEEK may be used in an IF statement or any other statement where a numeric value is accepted.

```
240 IF PEEK(J) = 49 THEN PRINT "YES"
```

Here is an interesting program using PEEKs and POKES. It doesn't do anything practical but it is a fun program. The program begins by filling the PET's screen with plus signs (ASCII 43). Then it randomly changes + signs to \* (asterisks) and if it finds an asterisk it is changed back into a + sign. Try it!



## TI AND TI\$ FUNCTIONS

In order to give you the ability to time operations in your programs the PET and CBM have provided a built-in clock that can be accessed using the TI and TI\$ functions.

### TI or Time Function

The TI function accesses an electronic counter that begins counting when the PET/CBM is first turned on. The increments are in 1/60 of a second, called "jiffies." Try entering the immediate mode command

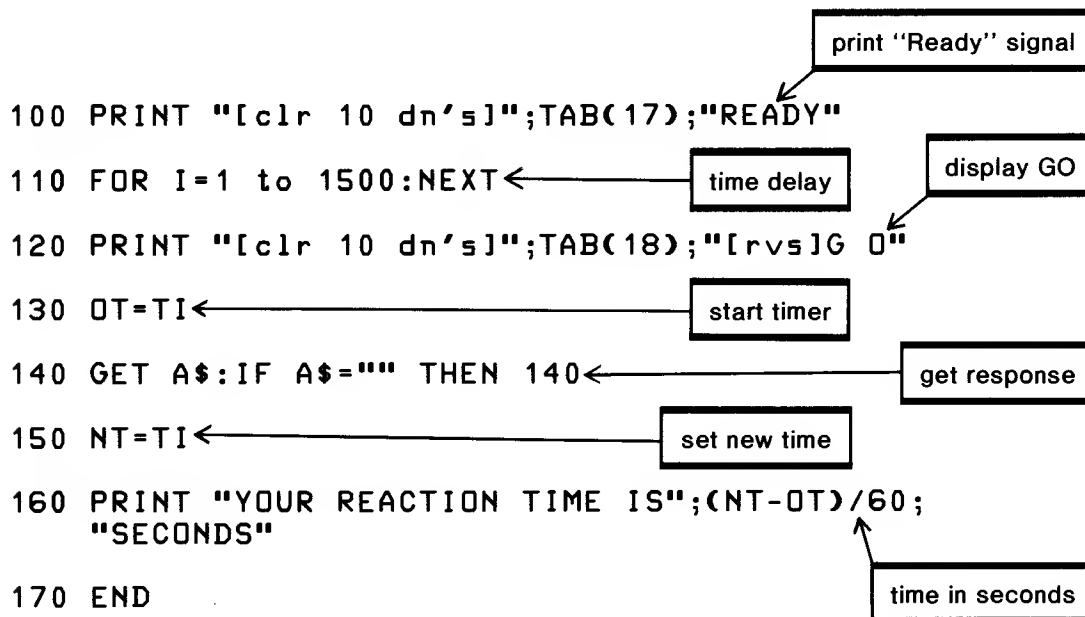
```
? TI
```

and you will get a number something like 67462, meaning it has been 67,462 jiffies since the computer was powered up. If you divide this number by 60 you get 1124.36667 or about 1124 seconds since the power was turned on. There are:

60 jiffies per second  
36,000 jiffies per minute  
216,000 jiffies per hour

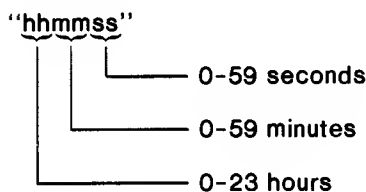
## Reaction Timer

How fast are your reactions? The following program is a simple reaction timer using the TI function that you can use to check your response time. Essentially the program takes the time prior to a GO signal and stores it in variable OT (Old Time). When you press any key, time is taken again and stored in NT (New Time). The reaction time is then the difference between these values divided by 60 to convert the time to seconds.



## TI\$ or Time\$

The TI\$ function returns the time in hours, minutes, and seconds in the form of a six-digit character string. The format of the string is:



Initially TI\$ is set to all zeros upon power-up of the computer. You may want it to reflect the actual time of day and this may be done by assigning the time to TI\$. This is done by using immediate mode as follows. Suppose we want to set TI\$ to 10:24 A.M. This would be entered a few seconds before the actual time has arrived.

```
TI$="102400"
```

Now wait until exactly 10:24 and then press return. The function now contains the time specified in the above expression. If you are entering time during the afternoon remember that this function records 24-hour time and adjust your figures accordingly.

Now try this brief program.

```
10 PRINT TIME$;"[1t 1t 1t 1t 1t 1t]";
20 GOTO 10
```

## DEF FN



The Define Function statement permits you to define your own functions for use in a BASIC program. Using DEF FN is a particular advantage when a formula needs to be used at several different places in the program. Rather than writing the formula each time it is needed you simply reference the function described at the beginning of the program.

The format of DEF FN is as follows:

DEF FNname(argument)=formula,

where

- name is the function name, consisting of one or two characters according to the rules of variable names.
- argument is a floating point variable that supplies a value for the function to operate upon.

For example, to define a function that calculates simple interest the formula  $i = prt$  is used, where

i is the interest  
p is the principal  
r is the rate  
t is the time in years

If we wanted to define a function to find the interest for a given investment the following code could be used.

```
10 DEF FNI(P)=P*R*T
20 R=0.185
30 T=1
40 PRINT FNI(5000)
```

Diagram annotations:

- Arrow from **Function I** to **FNI(P)**
- Arrow from **principal 'p' is an argument** to **P**
- Arrow from **18.5%** to **R**
- Arrow from **1 year** to **T**
- Arrow from **principal of \$5000** to **5000**

If a constant rate and time were used (rather unlikely in this case) the function could be defined as

```
10 DEF FNI(P)=P*0.185*1
```

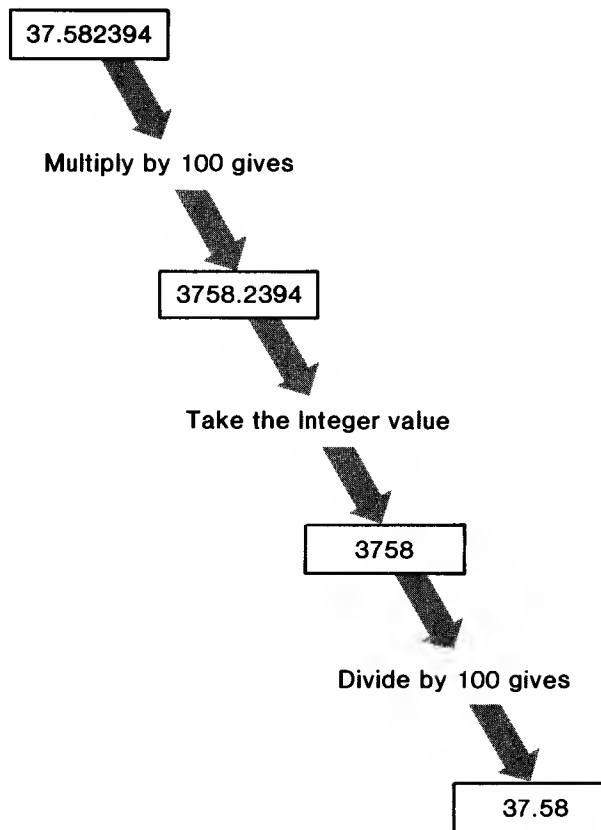
If the rate was constant but the time variable use

```
10 DEF FNI(P)=P*0.185*T
```

For another example consider the need to control the maximum number of decimal positions printed. In many situations the result of a calculation can have many decimal positions when only 1 or 2 are really necessary. In some BASICs a Print Using is available for this type of control but BASIC 4.0 doesn't have this feature.

One way of limiting a value to no more than two decimal positions is to multiply the number by 100 and take the integer portion of the result. Then divide this integer by 100 to get back to the original range of the value.

Example:



Here is the program to limit results to no more than 2 decimal positions.

```
100 DEF FNR(N)=INT(N*100)/100
110 FOR I=1 TO 3
120 READ N
130 PRINT FNR(N)
140 NEXT I
150 DATA 236.45678,1234.567,5.00123456
```

This program gives the output

```
236.45
1234.56
5
```

Several limitations are obvious from this output. One is that this function controls only the maximum number of decimal positions. Therefore in the case of the last value no decimals print. The other limitation is that the decimal points do not line up as would be required in reports such as those used for accounting applications.

## ARITHMETIC FUNCTIONS



Functions are subroutines that are already built into the ROM to let you do complicated arithmetic operations much more easily than if you had to write your own function or subroutine. Arithmetic functions use a function name and an argument that supplies the value to be acted upon.

For example, the integer function used in a Print statement is

```
PRINT INT(26.2987)
```

INT is the function name and 26.2987 is the argument. The function removes the fraction and returns the integer value of 26 to be printed.

Functions that return numeric values may be used any place a number is valid. Therefore they may be used in arithmetic statements, IF statements, FOR statements, subscripts, and even in other functions.

### ABS

The ABS function supplies the absolute value of the argument by removing its sign.

Examples:

```
10 N=ABS(256)
20 K=ABS(-256)
30 J=ABS(-23.05)
40 PRINT N,K,J
```

gives the output

```
256      256      23.05
```

### ATN

ATN returns the arctangent in radians of the argument. The returned value will be in the range  $\pm 17$ .

```
?ATN(30)           gives 1.53747533 radians
?180/π*ATN(45)      gives 88.72697 degrees
```

### COS

The COS function returns the cosine of the argument. The argument represents a value expressed in radians.

To find the cosine of .01 radians

```
?COS(.01)          gives .99995
```

The cosine of 1 radian is

```
?COS(1)            gives .540302306
```

The cosine of 90 degrees

```
10 R=90*π/180  
20 PRINT COS(R)
```

convert  
to radians

prints a cosine of 0.

### EXP

This function returns the value of  $e$  raised to the power of the argument ( $e^{\text{arg}}$ ) where  $e$  is the value 2.71828183. The argument must be in the range  $\pm 88.0296910$ .

```
?EXP(0)      gives 1  
?EXP(1)      gives 2.71828183  
?EXP(10)     gives 22026.4658
```

In the case of the EXP function an argument that exceeds the maximum will give an overflow error while an argument less than the minimum returns a 0 value.

### INT

The INT function returns the integer portion of a real number by truncating the fraction. The result is in real form, not integer(%). In the case of negative numbers INT adds 1 to the integer part after truncation.

```
?INT(27.57)   gives 27  
?INT(1.05)    gives 1  
?INT(0.05)    gives 0  
?INT(-27.57)  gives -28
```

### LOG

The LOG function returns the natural logarithm or log to the base  $e$ , where  $e$  is the value 2.71828183. If the argument is 0 or negative an Illegal Quantity message is generated.

```
?LOG(10)      gives 2.30258509  
?LOG(1)       gives 0  
?LOG(5000)    gives 8.5171932
```

To find log to the base 10 simply divide the natural logarithm by LOG(10). For example:

```
?LOG(40)/LOG(10) gives 1.60205999
```

### RND

The RND function generates random numbers between 0 and 1. These numbers can be of value in games and simulation programs. The function

```
RND(argument) returns a random number.  
RND(-argument) stores a new seed number for the generator.
```

Storing a new seed begins a new sequence of random numbers.

This step is useful when the PET is first turned on to ensure completely random numbers each time a program is run.

Now try the program

```
10 L=RND(-TI)
20 FOR I=1 TO 5
30 PRINT RND(1)
40 NEXT I
```

uses time function  
to store a seed value

This program displayed the sequence

```
.574960506
.0214301237
.558740495
.277825403
.0561438672
```

but if you run this program you will get a different sequence of random numbers.

### ***Creating a Specific Set of Random Numbers***

Often we have the need for a set of random numbers within a specific range. Take for example the range 0 to 9. This range can be created by multiplying the random number generated by 10 (1 larger than the maximum value required) and then taking the integer of this result.

```
20 N=INT(RND(1)*10)
```

This statement will generate a number in the range from 0 to 9. A number in the range 1 to 9 is created with the expression

```
20 N=INT(RND(1)*9+1)
```

The +1 adjusts the values 0 to 8 generated by the function to become 1 to 9, the range we wanted.

### ***SGN***

The SGN function is used to determine the sign of a number. If the number is

```
positive a + 1 is returned
zero     a  0 is returned
negative a - 1 is returned
```

Examples:

```
?SGN(25)    displays 1
?SGN(0)     displays 0
?SGN(-12)   displays -1
```

### ***SIN***

The SIN function returns the sine of the argument, which is expressed in radians.

Examples:

```
?SIN(.01)   displays 9.99983334E-03
?SIN(1)     displays .841470985
```



## **SQR**

The square root of a positive number is found with the SQR function.

Examples:

?SQR(36)	displays 6
?SQR(20.25)	displays 4.5
?SQR(4096)	displays 64

## **TAN**

TAN returns the tangent of the argument expressed in radians.

Examples:

?TAN(.01)	displays .0100003333
?TAN(1)	displays 1.55740772

## **Converting Radians to Degrees**

If the problem you are solving requires a value in degrees, rather than in radians, the following formula may be used.

$$\text{degrees} = 180/\pi * \text{radians}$$

for example

? 180/π \*ATN(30)

The reverse situation may be true when using functions like SIN, COS and TAN. To convert from degrees to radians use

$$\text{radians} = \text{degrees} * \pi / 180$$

for example

? SIN(60\*π/180)

## **STRING FUNCTIONS**

Functions in this category are like the arithmetic functions except that character strings are involved in the operation. A string function may require one, two, or even three arguments depending on the function. When more than one argument is used each argument is separated from the other with a comma.

## **ASC**

This function returns the ASCII code equivalent to the single character specified in the argument. ASCII codes are numeric values between 0 and 255 as listed in the Appendices.


?ASC("A")	displays 65
?ASC("1")	displays 49
?ASC("▲")	displays 223
A=ASC("\$")	assigns ▲ 36 to variable A

One of the uses for ASC is to translate characters that have been entered from the keyboard but are POKE'd onto the screen. In order to do this the keyboard characters must be changed to screen ASCII characters, which are a different ASCII sequence. The following is a subroutine that converts the variable B\$, which might have been read with

```
100 GET B$: IF B$ = "" THEN 100
110 GOSUB 7000
```

into the ASCII variable C which may then be POKE'd onto the screen.

```
7000 REM CONVERT B$ TO ASCII SCREEN CHARACTER C
7010 IF ASC(B$)<96 AND ASC(B$)>63 THEN
    C=ASC(B$)-64:GOTO 7050
7020 IF ASC(B$)<129 THEN C=ASC(B$):GOTO 7050
7030 IF ASC(B$)>159 AND ASC(B$)<192 THEN
    C=ASC(B$)-64
7040 IF ASC(B$)>191 THEN C=ASC(B$)-128
7050 RETURN
```

If the value A had been entered into B\$ this would be the ASCII value 65. Subroutine 7000 identifies this value and subtracts 64 from it in line 7010 creating the POKE value 1. The graphic symbol  is ASCII 223 and is converted to POKE value 95 in line 7040 by subtracting 128 from B\$ and storing the result in C.

### CHR\$

The CHR\$ function is the inverse of ASC in that it returns the character equivalent of the single ASCII code specified in the argument. The argument may be a numeric value from 0 to 255.

```
?CHR$("65")      displays A
?CHR$("49")      displays 1
A=233: ?CHR$(A)  displays 
```

One value of the CHR\$ function is its ability to display values that cannot be expressed in a Print statement. For example, the character (") is a delimiter and cannot be part of a character string in BASIC. It can be displayed with

```
100 PRINT CHR$(34)
```

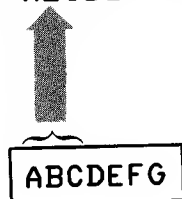
### LEFT\$

The LEFT\$ function is used to extract the leftmost characters from a string. This function requires two arguments:

LEFT\$(string, length)  
 —string is the character string to be used in the operation.  
 —length is the length of the string to be extracted and must be in the range 0 to 255.

Example:

```
?LEFT$("ABCDEFG",3)  displays ABC
```



A common use for LEFT\$ is to select the first letter of a user's response. For example, if a YES answer is expected to a query we may examine only the first byte of the response for a "Y". This minimizes processing errors due to spelling mistakes, carelessness, or simply not entering the complete word by the operator of the program.

```
100 INPUT R$
110 IF LEFT$(R$,1)="Y" THEN 200
```

Here is another example:

```
10 T$="[rt rt rt rt rt rt]"
20 INPUT K
30 PRINT LEFT$(T$,K):"WORDS"
```

This code will accept a value K that determines how many cursor rights are used before printing the value WORDS. This technique, or variations of it, can be useful to control the amount of spacing prior to printing a value. Of course other cursor controls or even other values may be substituted to be selected prior to printing.

### LEN

The LEN function returns the length of the string argument supplied. For example:

```
?LEN("STRING")← displays 6
10 X$="TO THY OWN SELF BE TRUE"
20 PRINT LEN(X$)← displays 23
```

### MID\$

The MID\$ function extracts any specified portion of the character string identified in the argument. The arguments specify the string, the position to begin extraction, and the number of characters to be extracted.

MID\$(string, position, length)

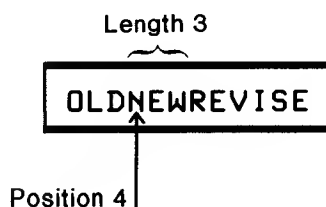
—string is the character string to be used in the operation.

—position is the position of the first character to be extracted from the string. This value must be from 1 to 255.

—length is the number of characters to be extracted, from 1 to 255.

Example:

```
10 A$="OLDNEWREVISE"
20 PRINT MID$(A$,4,3)← displays NEW
```



Suppose we want to input a word and print it diagonally across the screen. Try the following code.

```
10 INPUT A$
```

```
20 FOR I=1 TO LEN(A$)
```

```
30 PRINT "[dn rt]";MID$(A$,I,1);
```

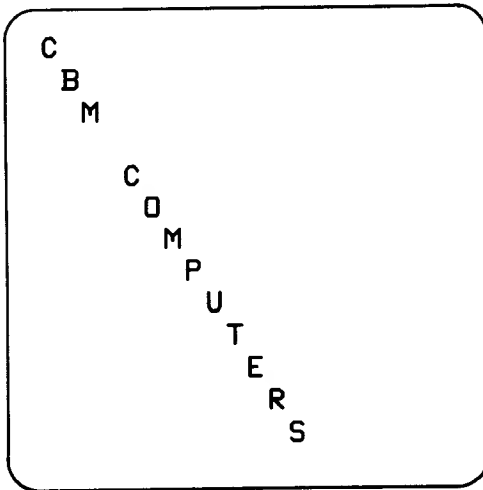
```
40 NEXT I
```

will be 13 for the sample data

selects each letter  
from A\$

move down 1 line and leave  
one space for each letter

If you enter the input CBM COMPUTERS, the following output will be displayed:



```
C
 B
  M
   C
    O
     M
      P
       U
        T
         E
          R
           S
```

MID\$ is useful in applications where the program needs to look through a string for a specific value. For instance, in a CAI application the student might be required to answer a question such as:

**WHAT ARE TITAN AND DIONE ?**

The response to this might be

**TITAN AND DIONE ARE MOONS OF SATURN.**

This response might be processed by the following code, which looks for the word MOONS as part of a correct response.

```
100 INPUT X$
100 FOR I=1 TO LEN(X$)-4
120 IF MID$(X$,I,5)="MOONS" THEN 200
130 NEXT I
```

The FOR loop in this code moves through the X\$ string one character at a time looking at groups of 5 characters for the string "MOONS". If this is found the program branches to 200 where a positive message might be printed or the program could then look for the word "SATURN" to see if the student associated the moons with the planet Saturn. In any event the program is looking only for the occurrence of these words and in no way analyzes the context in which they have been used.

## RIGHT\$

RIGHT\$ is similar to the LEFT\$ function except that it extracts the rightmost characters from the string specified in the argument.

RIGHT\$(string, length)

—string is the character string to be used in the operation.

—length is the length of the string to be extracted and must be in the range 1 to 255.

?RIGHT\$("ABCDEFG", 3) ← displays EFG

10 A\$=RIGHT\$(B\$, 7) ← assigns the 7 rightmost characters of B\$ to A\$

## STR\$

The STR\$ function returns the equivalent string value of the numeric argument. This operation may be useful when a numeric result needs to be combined with a character string or used in a string operation.

?STR\$(278.05) ← displays 278.05

10 N=125.78  
20 K\$=STR\$(N) ← converts N to K\$

30 PRINT RIGHT\$(K\$, 3) ← displays .78

## VAL

VAL is the reverse of STR\$ since it returns the numeric equivalent of the string argument.

?VAL("80.175") ← displays 80.175

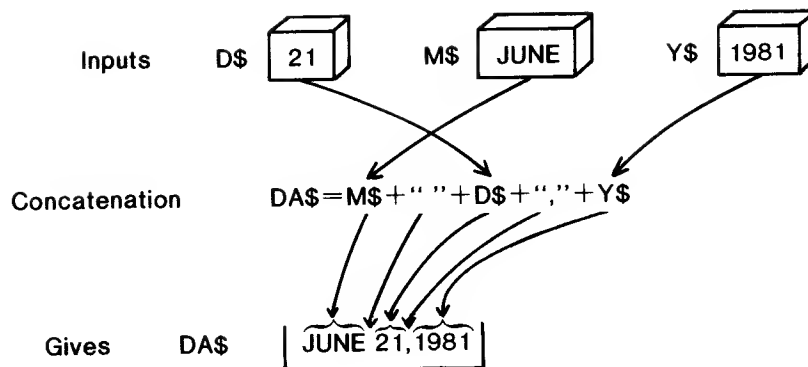
10 C\$="THE ANSWER IS -75"  
20 A\$=RIGHT\$(C\$, 3)  
30 PRINT VAL(A\$) ← displays -75

## CONCATENATION

Although not a function, concatenation is a useful string operation that combines two or more strings to form a single string of characters. The + sign is used as the concatenation operator but should not be confused with the arithmetic plus sign. Although the same character is used the results are quite different.

? "CONCAT" + "ENATION" ← displays CONCATENATION

```
10 INPUT "DAY";D$
20 INPUT "MONTH";M$
30 INPUT "YEAR";Y$
40 DA$=M$+" "+D$+" ", "+Y$
50 PRINT DA$
```



An interesting change to this program in statement 50 concatenates the reverse control character to `DA$` as follows:

```
50 PRINT "[rvs]" + DA$
```

This simple change causes all of the contents of `DA$` (the date) to be printed in reverse characters.

## PLOTTING GRAPHS

Drawing graphs on the PET presents an interesting challenge for the programmer. Normally if a graph is displayed as its values are calculated it will come out sideways on the screen and you will need to twist your head sideways to read it. Using a technique developed by Robert Barrett ("Creative Computing", April 1979) the program that follows displays correctly and is much easier to read. This shift is made by first storing the points of the graph in an array and then displaying them horizontally.

For example, if the function  $\text{SIN}(X)$  is computed for values of  $X$  between 0 and 180 (in degrees) the program would produce the output shown in figure 6.7.

Figure 6.8 is the flowchart for the solution to this problem and figure 6.9 shows the program. Notice that lines 140 to 160 in the program show three different functions. Currently 150 and 160 are Remark statements and so they do not affect the running of the program. To use another function it is necessary to change line 140 to a Remark and remove the REM from the function you want to try.

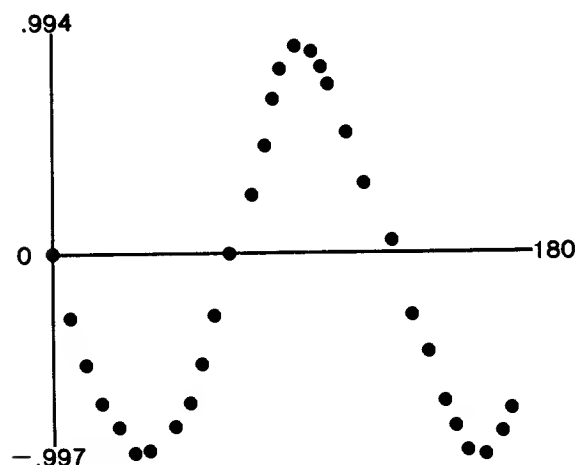


Figure 6.7 Plotting  $\text{SIN}(X)$  between 0 and 180

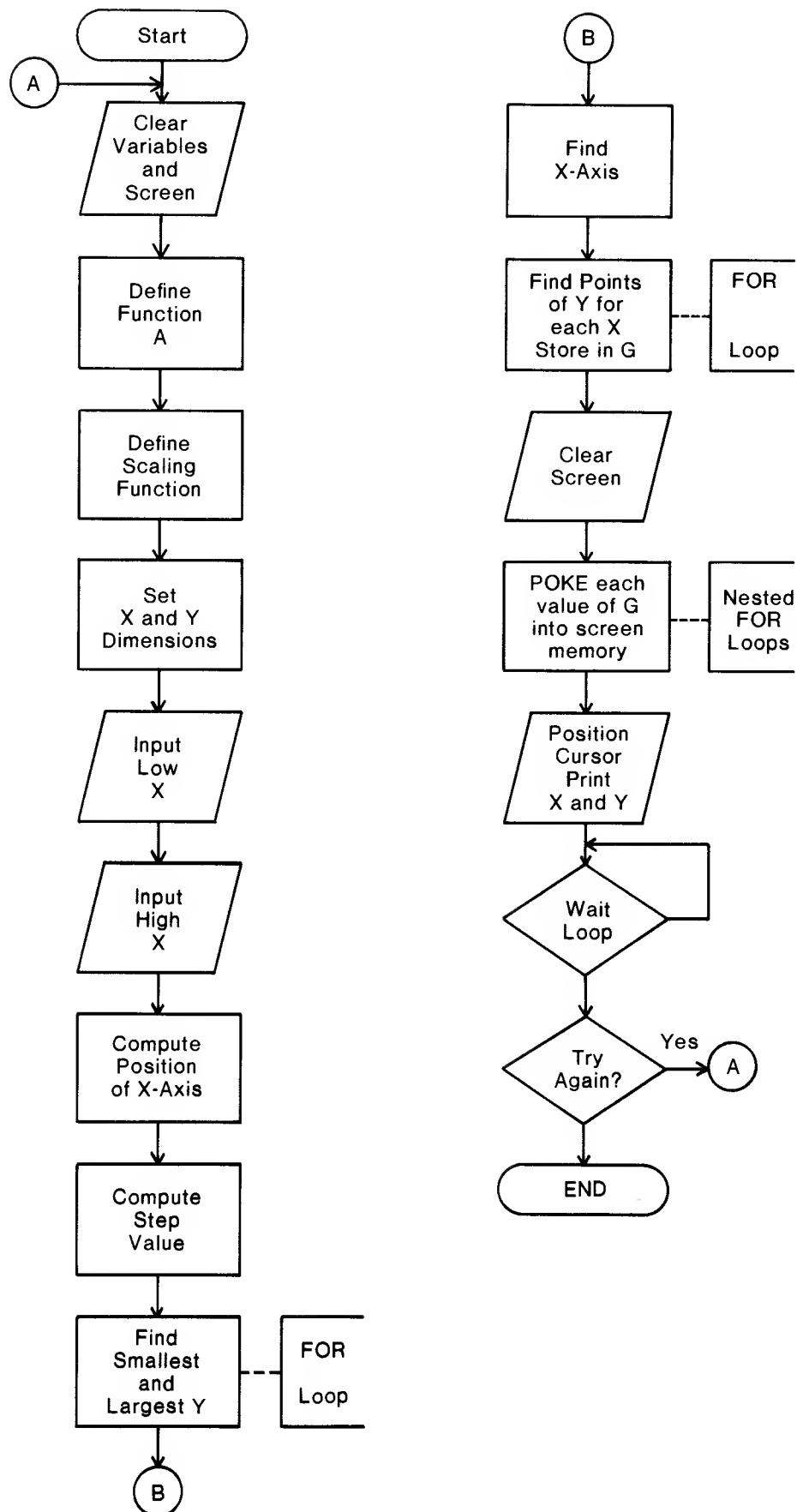


Figure 6.8 Plotting a graph flowchart

```

100 REM PLOTTING GRAPHS
110 CLR←clear variables
120 PRINT"[clr] "
130 DIM G(40,25)
140 DEF FNA(X)=SIN(X)
150 REM DEF FNA(X)=COS(X/20)}try these functions
160 REM DEF FNA(X)=(X2)by removing REM
170 DEF FNS(X)=INT((X-M)/(S-M)*YA+1.5)←scaling function
180 YA=22:XA=30←sets the Y-axis
190 INPUT "LOW VALUE OF X";XLand X-axis
200 INPUT "HIGH VALUE OF X";XH
210 PRINT"[dn rvs]FINDING MAXIMUM SCALE[off]"
220 Y1=INT(-XL/(XH-XL)*XA+1)←position of Y-axis
230 D=(XH-XL)/XA
240 FOR X=XL TO XH STEP D
250 Y=FNA(X)
260 IF S<Y THEN S=Y
270 IF M>Y THEN M=Y
280 NEXT X
290 I=0
300 X1=FNS(0)
310 PRINT"COMPUTING GRAPH POINTS"
320 FOR X=XL TO XH STEP D←find Y values between the
330 Y=FNS(FNA(X))←range of X (low-high)
340 I=I+1
350 G(I,X1)=-1
360 G(I,Y)=1
370 NEXT X
380 PRINT"[clr] "
390 P=32767
400 FOR I=25 TO 1 STEP-1
410 P=P+3
420 FOR J=1 TO 37
430 P=P+1
440 IF G(J,I)=-1 THEN POKE P,64←horizontal line
450 IF J=Y1 THEN POKE P,66←vertical line
460 IF G(J,I)=1 THEN POKE P,81←white dot
470 NEXT J
480 NEXT I
490 PRINT "[home]";LEFT$(STR$(S),5)
500 PRINT "[home 12 dn's]";XL;(S),5)
510 PRINT TAB(35);XH
520 PRINT"[10 dn's]";LEFT$(STR$(M),5);
530 GETA$:IF A$="" THEN 530
540 INPUT " TRY ANOTHER SET OF VALUES Y[3 left]";A$
550 IF A$="Y" THEN 110

```

Figure 6.9 Program to plot graphs (for PET and CBM 4000 series only)



Function S in line 170 is a scaling function which has the purpose of scaling each value of Y to fit within the range of positions available for plotting the graph. The value of X can vary widely so line 220 computes the position of the Y-axis along the X-axis.

Lines 240 to 280 compute values of Y to determine minimum and maximum for the purpose of scaling the final results. Next, lines 310 to 370 compute the values of Y and store them in the array G at the appropriate intersection of X and Y. Finally, lines 400 to 520 plot the graph and print its coordinates using a combination of POKEs and Print statements.

## CONTROLLING DECIMAL POSITIONS

Earlier in this chapter we saw how to limit the number of decimal positions to no more than two. However, a number with one or even no decimal positions would not align exactly with numbers containing two or more decimal positions. This means the decimal points will not align in a straight column, which may be desirable for some applications such as accounting or financial reports.

The technique discussed here will limit all decimal numbers to two decimal positions regardless of the number of decimals in the original number. With a slight modification to the program the results could control output to any number of decimals required. Figure 6.10 shows a variety of decimal numbers and indicates the results required of the program.

Original Number	Desired Results
1	1.00
2.5	2.50
125	125.00
.05	0.05
.125864	0.12
79.16	79.16
-14.5	-14.50
-.458	-0.45

Figure 6.10 Editing to two decimal places

In this example, note that every number results in two and only two decimal positions. In some cases, such as the value 1, two decimal positions (.00) are appended to the number, while in other cases (.125864) decimal positions are truncated to result in two decimals.

Figure 6.11 shows the steps followed to take a number and adjust its decimal positions to two. First the number is separated into the integer (left) and fractional (right) parts. The fractional part is then adjusted to two digits, adding a decimal point if necessary and then concatenated back onto the integer part to create the final result.

Figure 6.12 shows the flowchart for this problem and figure 6.13 is the complete program. In the program statements 200 to 330 are the subroutine while statements 100 to 190 are a driver routine.

### Driver Routines

A driver routine is often used to test a part of a program without the need for having the complete program in the computer at the time of testing. For instance, if we were writing a complete Accounts Receivable program the decimal subroutine would be required. Using a driver initially lets us try the solution of limiting decimal points for a variety of values without the concern for the hundreds of other statements the full-scale program may eventually have. When we are satisfied with the solution for this subroutine it may be combined with the full program.

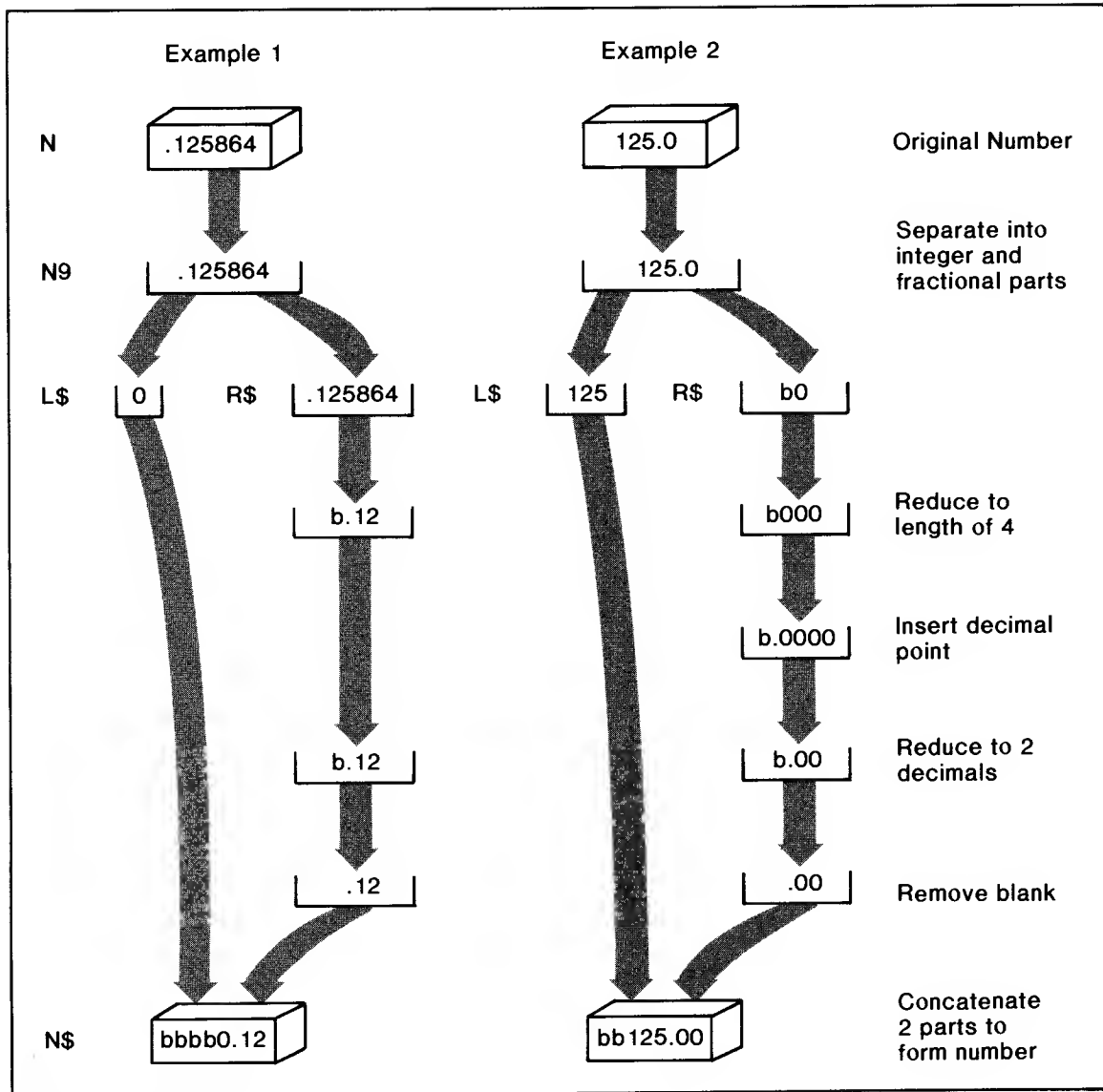


Figure 6.11 Converting to two decimal places

## CAI CHESS AND PROGRAM GENERALIZATION

Computer Assisted Instruction is one application area in education that the computer made possible. Although the program discussed here presents some initial instruction in chess it is generalized to the extent that by changing the data other subject matter may be presented in a CAI format. Figure 6.14 shows the arrangement of the data for this program.

The approach taken here is to develop the program logic in such a way that it is essentially independent of the subject to be presented. Rather than using a statement such as

```
100 PRINT "A RANK IS A HORIZONTAL ROW OF SQUARES"
```

the string is placed in a data statement and then read and printed.

Although this approach at first may seem more involved it actually means a lot less work in the long run. The reason for this is that only one Read and Print is necessary regardless of the amount of data to be read. Therefore the CAI instruction may be as long as needed without lengthening the program itself. Only the data length is increased.

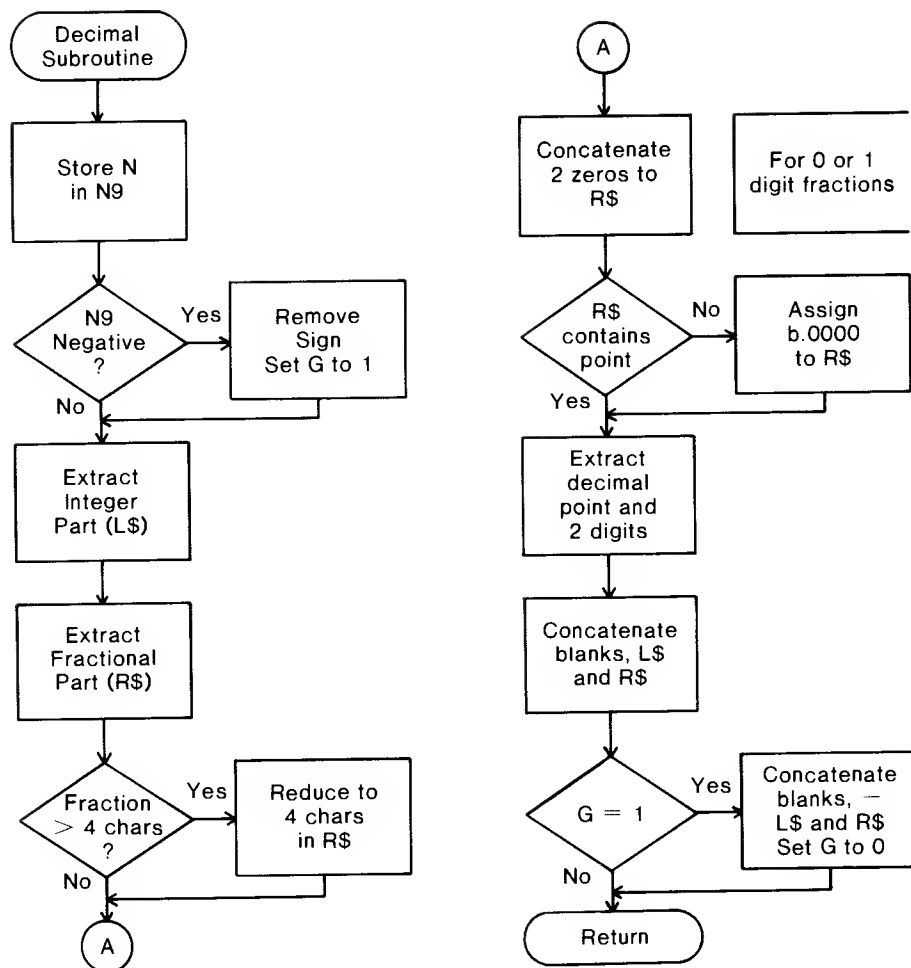


Figure 6.12 Flowchart for converting to two decimals

```

100 REM TEST DRIVER FOR DECIMAL NUMBERS
110 REM SUBROUTINE
120 FOR I=1 TO 8
130 READ N
140 PRINT N,
150 GOSUB 210
160 PRINT N$
170 NEXT I
180 STOP
190 DATA 1,2.5,125.0,.05,.125864,79.16,-14.5,-.458
200 REM
210 REM DECIMAL CONVERSION SUBROUTINE
220 N9=N
230 IF N9<0 THEN N9=ABS(N9):G=1
240 L$=STR$(INT(N9))
250 R$=STR$(N9-INT(N9))
260 IF LEN(R$)>4 THEN R$=LEFT$(R$,4)
270 R$=R$+"00"
280 IF R$=" 000" THEN R$=" .0000"
290 R$=LEFT$(R$,4)
300 R$=RIGHT$(R$,3)
310 N$=RIGHT$(" "+L$+R$,8)
320 IF G=1 THEN N$=RIGHT$(" -"+L$+R$,8):G=0
330 RETURN
  
```

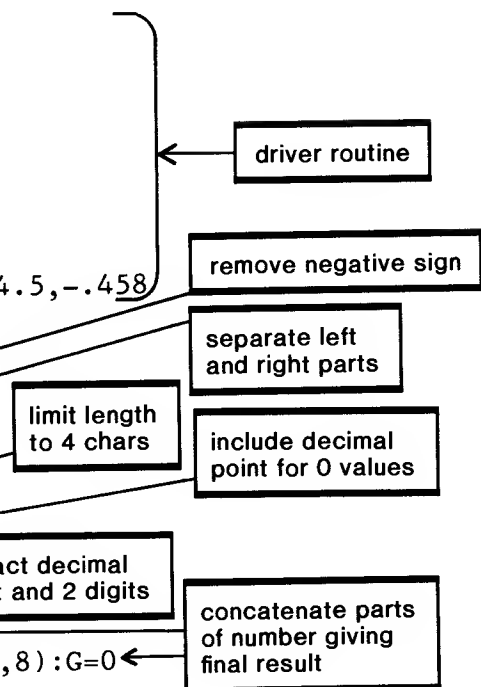


Figure 6.13 Program for converting to two decimals

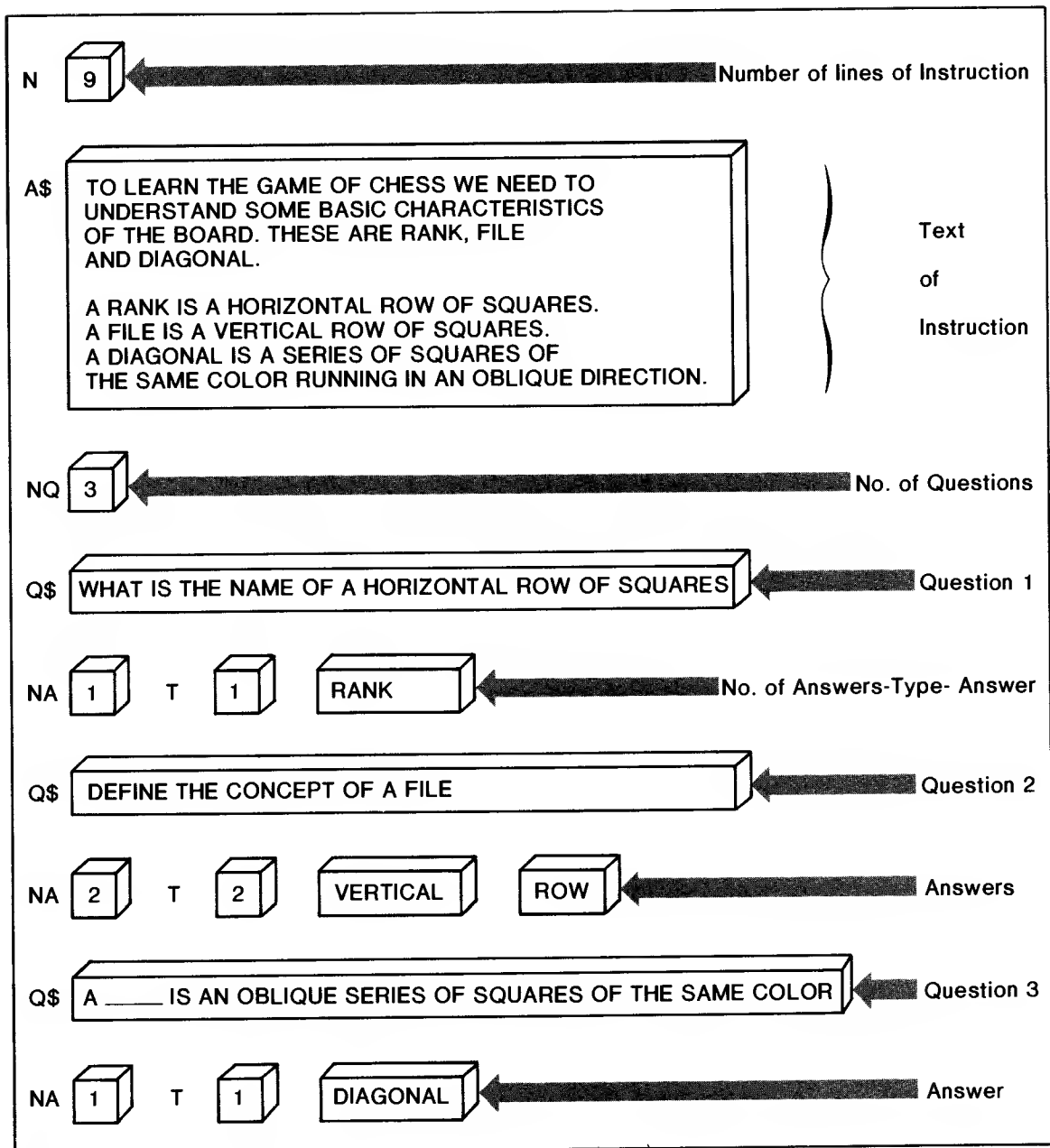


Figure 6.14 One data set for CAI chess

### The Data

Each set of data begins with a numeric value (called N in the program) that indicates the number of lines of instruction to follow. This value is 9 in the sample set of data indicating there are 9 lines of instruction including a blank line. The actual lines of text then follow in subsequent data statements to be read into A\$ by the program.

Following the text material is another number that tells the program how many questions have been supplied for this text. In the example the value is 3 and will be followed by a set of three questions and their answers. The 3 representing the number of questions is read into the variable NQ.

Now we have the first question:

WHAT IS THE NAME OF A HORIZONTAL ROW OF SQUARES?

This question is read into Q\$ and is followed by a 1 which defines the number of answers (NA). Next is a number that defines the type of answer to follow. There are three different types:

- 1—Single answer
- 2—AND type answer
- 3—OR type answer

The first type requires a single answer that could be either a word or phrase. In the first question the answer is RANK. The program will search the answer given by the user to determine if it contains the word RANK. If it does the answer is considered to be correct.

A type-2 answer indicates that more than one word is required in the answer and that each word must be present. In the second question:

#### DEFINE THE CONCEPT OF A FILE?

both words VERTICAL and ROW must be present in the answer for it to be correct. The logic of this evaluation is based on the Boolean 'and' truth table.

Type 3 is based on the Boolean 'or' and therefore requires that only one of several possible answers be submitted. The question:

#### NAME A MAN THAT HAS ONLY TWO PIECES?

has three possible correct answers. ROOK, BISHOP, or KNIGHT are the possibilities. If any one of these answers is given, the response is considered correct. The answer given could be in the form of a single word such as:

#### BISHOP

or it could be a complete sentence. For example:

#### A ROOK IS A MAN WITH 2 PIECES.

In either case the program would scan the answer for the correct word within the answer string.

### ***The Program***

**Random Responses** Programs used for CAI typically need to respond to both correct and incorrect answers from the user. Of course this can be a simple matter of displaying "correct" or "incorrect" but this tends to become monotonous after a few answers. To add a little variety and spice to your answers it is possible to use the random function to select one response from a collection of responses.

Figure 6.15 shows how a set of 4 correct answers are set up in array CR\$ to be selected for a response to a correct answer. The random function generates a value between 1 and 4, which then acts as a subscript to extract the response from the array.

**Instring Search** Some languages have a statement with the capability of searching a string variable for the presence of a particular word or set of characters. This is called an Instring Search. Since PET/CBM BASIC does not have this feature it is necessary to write a subroutine that can accomplish the same type of search. The need for this approach becomes apparent when we consider the types of responses we are likely to get from the user of this program. Rather than single-word answers the user can respond with a sentence, which is a more natural means of communication or, if preferred, with a single word or phrase.

Figure 6.16 shows how to implement a search for the string IN\$ within the longer string A\$. If IN\$ is found the Found Switch (FS) is set to 1 indicating the presence of IN\$ in A\$. Otherwise the string is not found and FS remains a 0 value.

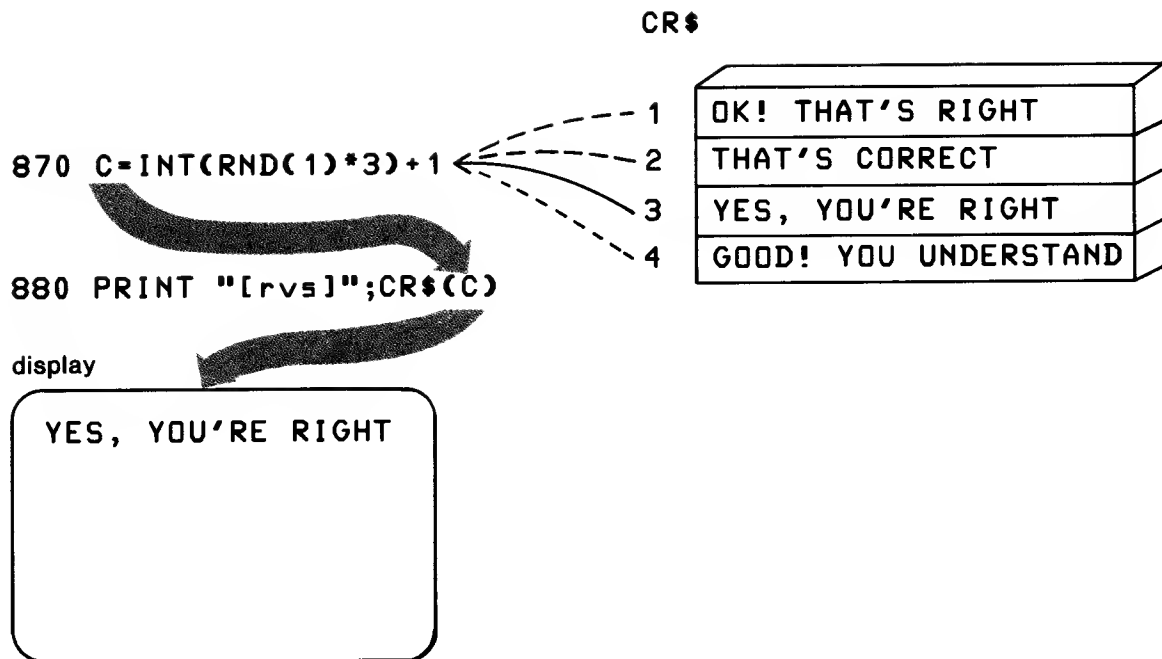


Figure 6.15 Selecting a random correct response

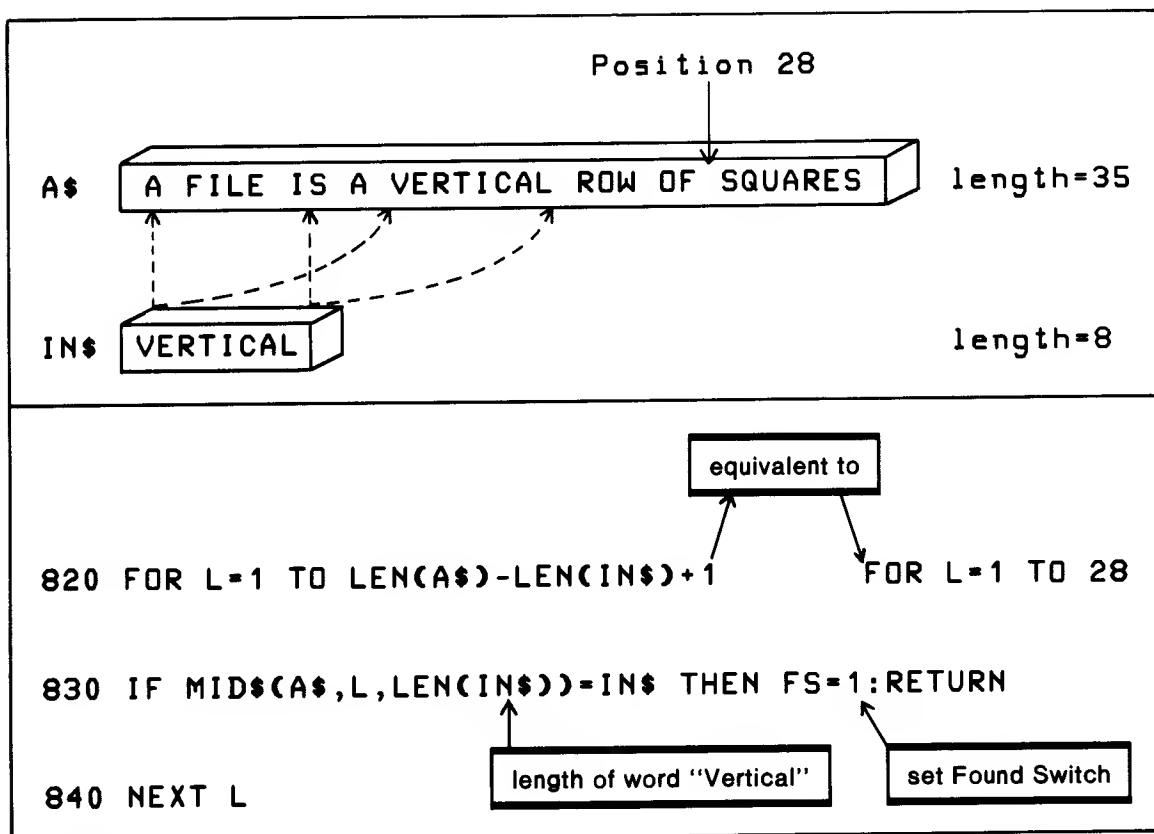


Figure 6.16 Instrng Search subroutine

**Checking AND Type Responses** When an AND type of response is required it means there are two or more words expected in the answer. The answer string will be examined for the presence of each of these words and if all of the words are found the answer is considered correct. Figure 6.17 demonstrates the solution.

The array A1\$ contains each of the words to be searched for in the input string using subroutine 810 for the Instring Search. For each match found (FS = 1) the search moves on to look at the next word at A1\$(M). If the word was not found in the input string (FS = 0) then subroutine 910 is called to produce the incorrect answer response. Finally, if all words are found in the input string the loop terminates and subroutine 860 produces the correct answer response.

Basically the OR type of response functions in the same way, the primary difference being that only one of the words in array A1\$ needs to be found in the answer string.

The complete program for Computer Assisted Instruction on the game of chess is shown in figure 6.18. Additional instructional material may be added to the DATA statements to provide a continuation of the lesson.

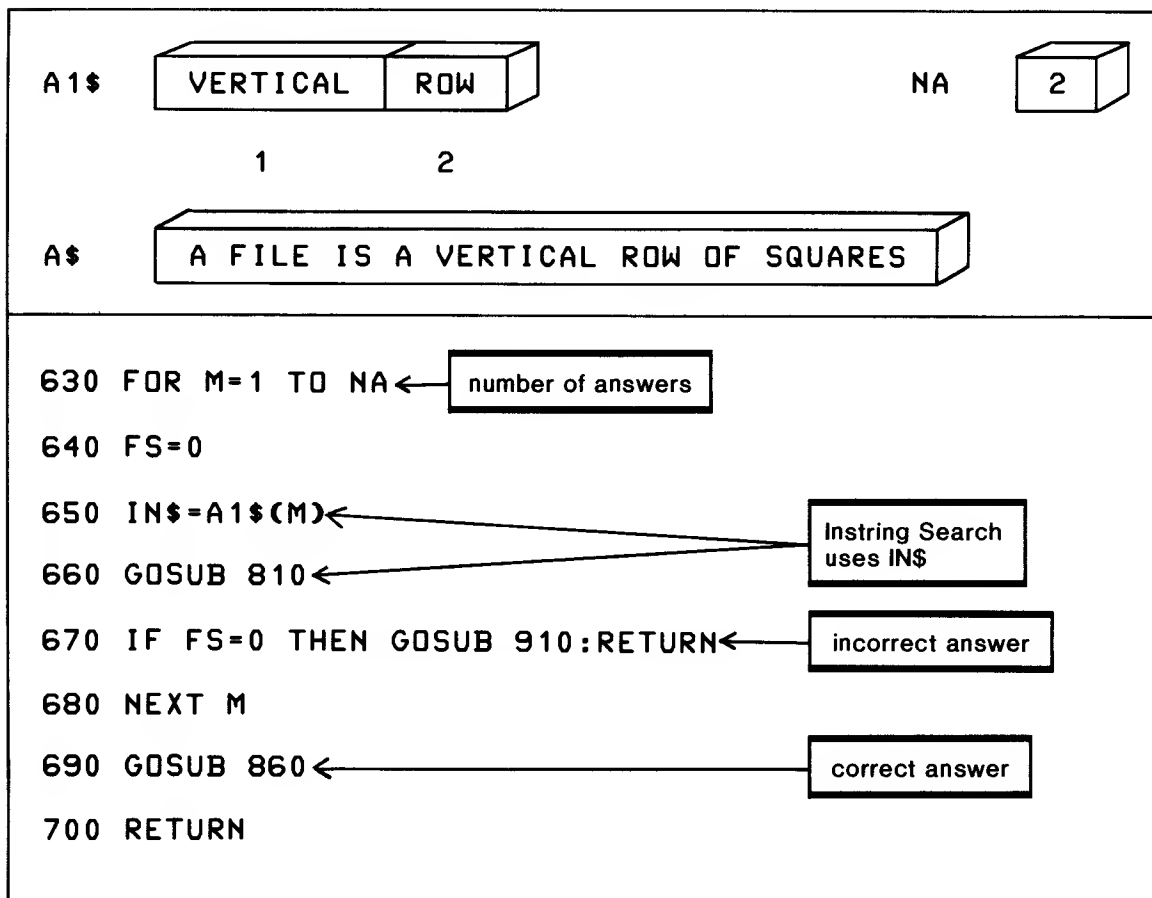


Figure 6.17 Checking AND type responses

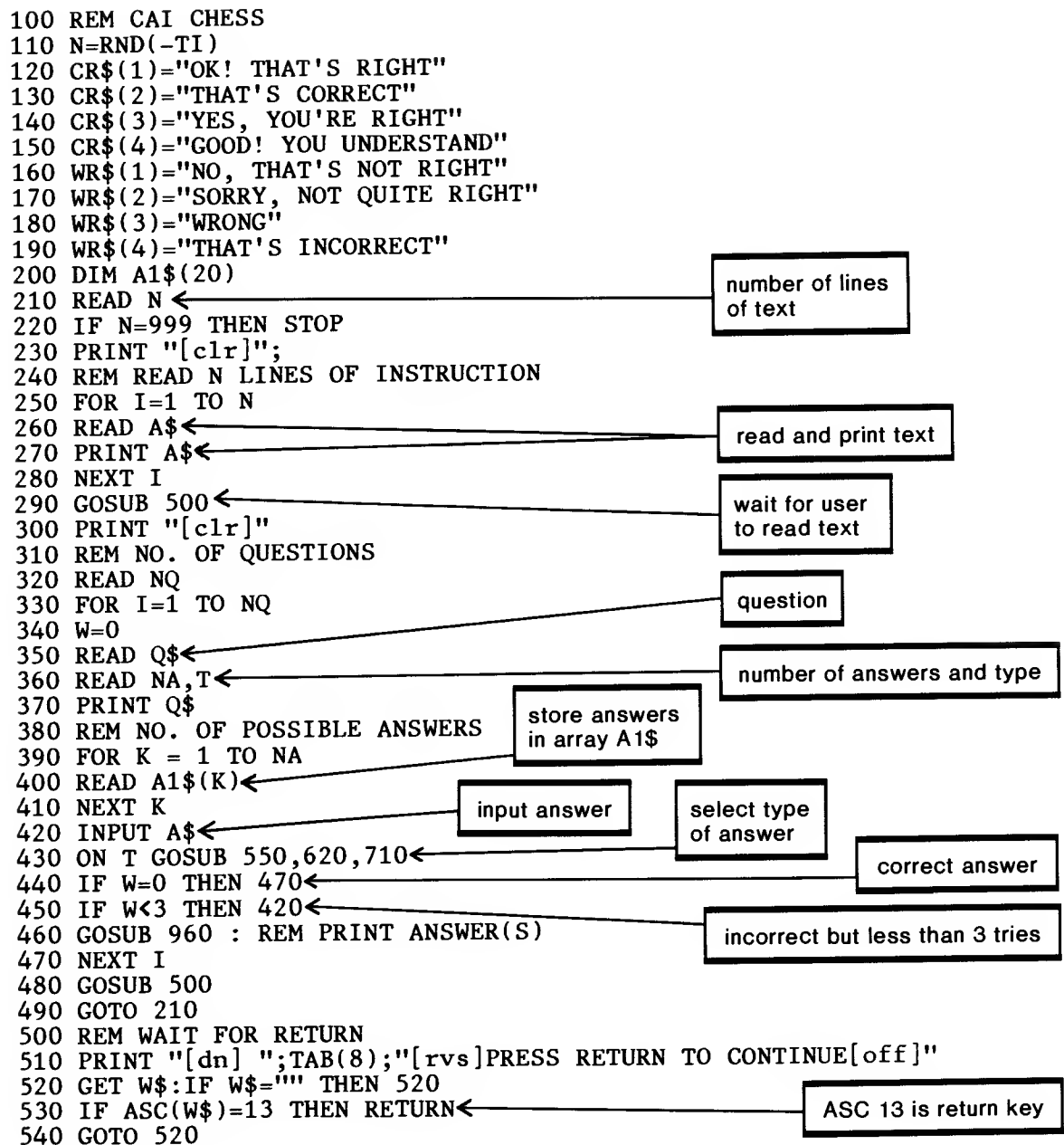
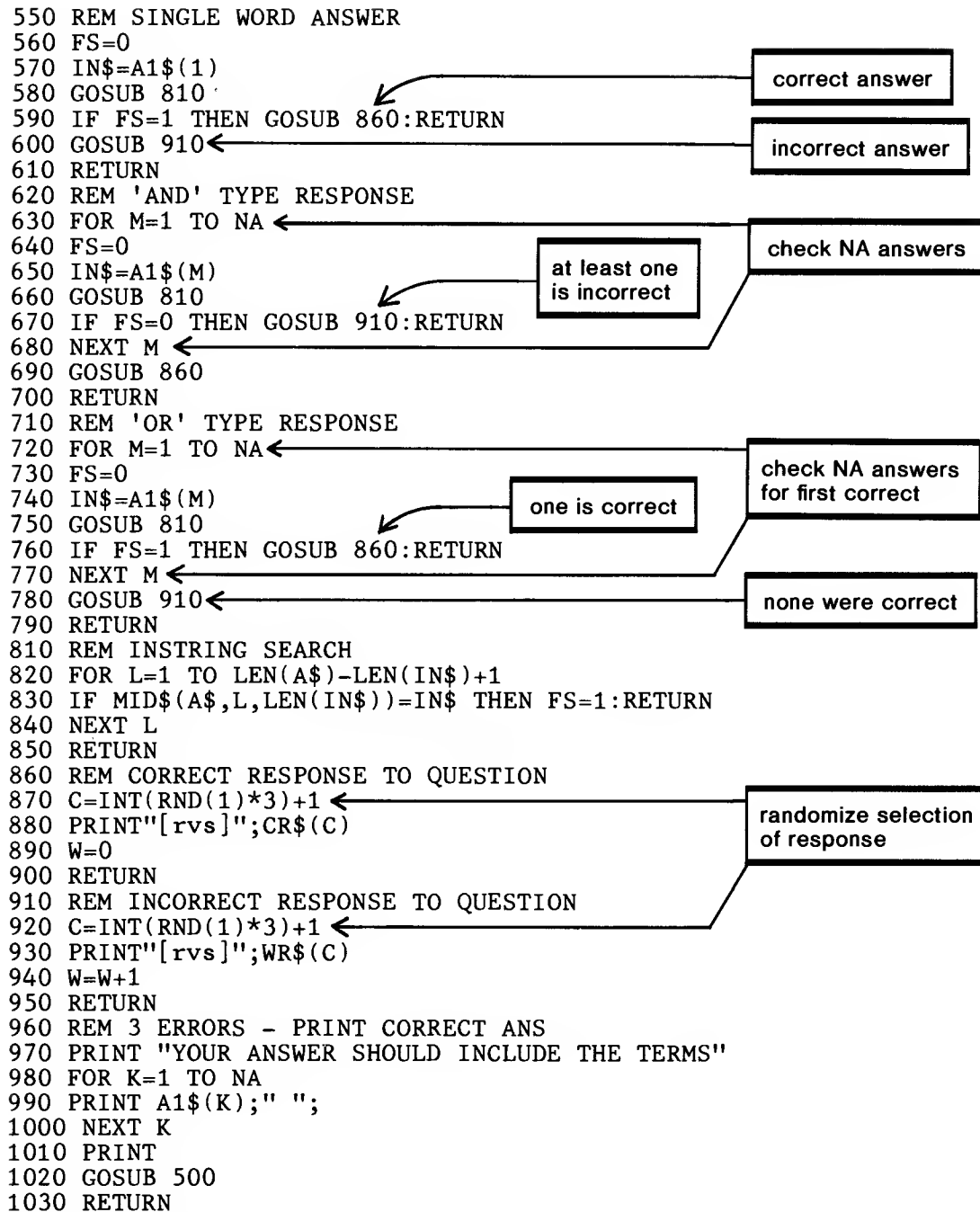


Figure 6.18 CAI chess program





**Figure 6.18** CAI chess program (continued)

```

1040 DATA 9
1050 DATA "TO LEARN THE GAME OF CHESS WE NEED TO"
1060 DATA "UNDERSTAND SOME BASIC CHARACTERISTICS"
1070 DATA "OF THE BOARD. THESE ARE RANK, FILE"
1080 DATA "AND DIAGONAL."
1090 DATA " "
1100 DATA "A [rvs]RANK[off] IS A HORIZONTAL ROW OF SQUARES."
1110 DATA "A [rvs]FILE[off] IS A VERTICAL ROW OF SQUARES."
1120 DATA "A [rvs]DIAGONAL[off] IS A SERIES OF SQUARES OF"
1130 DATA "THE SAME COLOR RUNNING IN AN OBLIQUE DIRECTION."
1140 DATA 3
1150 DATA "WHAT IS THE NAME OF A HORIZONTAL ROW OF SQUARES"
1160 DATA 1,1,"RANK"
1170 DATA "DEFINE THE CONCEPT OF A FILE"
1180 DATA 2,2,"VERTICAL","ROW"
1190 DATA "A rrrrr IS AN OBLIQUE SERIES OF SQUARES OF THE SAME COLOR"
1200 DATA 1,1,"DIAGONAL"
1210 DATA 10
1220 DATA "THE GAME OF CHESS USES 16 BLACK AND"
1230 DATA "16 WHITE PIECES WITH THE FOLLOWING"
1240 DATA "NAMES AND DISTRIBUTION."
1250 DATA " "
1260 DATA "          1 - KING"
1270 DATA "          1 - QUEEN"
1280 DATA "          2 - ROOKS"
1290 DATA "          2 - BISHOPS"
1300 DATA "          2 - KNIGHTS"
1310 DATA "          8 - PAWNS"
1320 DATA 6
1330 DATA "HOW MANY BLACK PIECES ARE USED IN CHESS"
1340 DATA 1,1,"16"
1350 DATA "NAME EACH PIECE.(TYPE EACH NAME FOLLOWED BY A SPACE)"
1360 DATA 6,2,"KING","QUEEN","ROOK","BISHOP","KNIGHT","PAWN"
1370 DATA "HOW MANY PIECES ARE WHITE KINGS"
1380 DATA 1,1,"1"
1390 DATA "HOW MANY PIECES ARE BLACK QUEENS"
1400 DATA 1,1,"1"
1410 DATA "NAME A MAN THAT HAS ONLY TWO PIECES"
1420 DATA 3,3,"ROOK","BISHOP","KNIGHT"
1430 DATA "HOW MANY WHITE PAWNS ARE THERE"
1440 DATA 1,1,"8"
1450 DATA 999,"END-OF-DATA"

```

additional material  
may be inserted here

renumber end line

Figure 6.18 CAI chess program (continued)

## **REVIEW QUESTIONS—CHAPTER 6**

1. What is the benefit of using a GET statement instead of the INPUT statement? Why is a loop necessary with the GET?
2. What are some of the limitations of the GET compared to the INPUT statement?
3. Consider how you would write a module to get more than one character of input using the GET statement. How would you handle corrections such as using the delete or cursor keys?
4. What are some of the benefits of using the ON group of statements? Discuss the pros and cons of ON—GOSUB versus ON—GOTO.
5. Describe the purpose of the POKE instruction. What is the only type of data that may be POKE'd?
6. Write some BASIC code using POKES that will draw a (1) horizontal line, (2) vertical line.
7. What does the PEEK statement do? How is it different from POKE?
8. Write a small program to cause a circle to move slowly across the screen from left to right.
9. What is a function? Name some. In general what is the difference between an arithmetic function and a string function? Give a specific example.
10. Driver routines are sometimes used in programming. What is a driver routine and how is it used?
11. Using the CAI chess program as a model write a program to give instruction in an interactive mode on a subject you are familiar with.

# **7**

## ***Interacting with the User of Your Program***

**A**s programmers of microcomputers we may frequently write programs that are intended for other than our own use. Rather the programs we develop are directed to a student audience, to our colleagues, or for use by others in our home or business. It is therefore crucial that we develop programs in such a way that they interact with the user at a suitable intellectual level and with an effectiveness that permits ease and simplicity of operation.

The intellectual level considers both the language and technical ability of the program's user and employs prompting and/or dialogue that is consistent with this level. Effectiveness of operation considers the methods used to communicate with the user. These methods may include queries, prompting, codes, menus, form filling, and so on. Suitable choices often determine a person's success in using the program and his willingness to use it again.

### ***USER LEVEL***

When we develop a program it is usually intended for a specific audience. In general, we should assess the age, education, training, intelligence, and motivation of the user. After a general assessment it is useful to place the user in one of the following three categories:

#### ***Casual***

This is a user who uses a microcomputer infrequently and generally has no training in computers. All first-time users fall into this category. So do many students and educators.

#### ***Trained***

A trained user is a person who has been given formal (sometimes informal) training on the use of computers. This training might be limited to the use of a particular program or be as broad as a computer literacy course.

#### ***Programming Skills***

This is the most sophisticated user. He or she will have done some programming and be familiar with programming terminology and language syntax. Since you have been studying programming in this book you fall into this category.

In addition to these three categories a user may operate the program on two different levels. Noted computer consultant James Martin defines these levels as an active or passive operator. An active operator is one who initiates program action by entering commands. This level is typical of games where the user enters values which might control the direction and speed of a ball on the screen.

A passive operator is one who takes action based on the program's initiative—for instance, when a program asks for the user's name. Many programs will use both active and passive interaction.

## **USER DIALOGUES**

### **Prompting**

In a sense any information that requires a user response represents prompting. However, our interest here is when the program displays a statement or question and then waits for the user to type a response. Prompting is appropriate for all levels of users but the language of the prompt should be directed to the specific user level. In any case, courtesy should be used. A prompt such as

**PLEASE ENTER YOUR NAME ?**

is much better than a curt

**NAME ?**

This kind of prompt is usually implemented as follows:

```
100 PRINT "PLEASE ENTER YOUR NAME";  
110 INPUT N$
```

or more directly and efficiently as

```
100 INPUT "PLEASE ENTER YOUR NAME";N$
```

With this type of prompt you may get a variety of responses, such as:

```
JOHN  
JOHN SMITH  
SMITH
```

In some programs this doesn't matter but if it does make a difference the prompt should be more specific.

**PLEASE ENTER YOUR SURNAME?**

Some prompts may have simple alternatives such as YES or NO, TRUE or FALSE, ADD or SUBTRACT. In these situations the prompt should indicate which responses are expected.

```
100 INPUT "DO YOU WANT MULTIPLE CHOICE(YES/NO)";A$  
110 IF A$="YES" THEN 200
```

This prompt indicates clearly that a YES or NO answer is expected. Some programs also permit the user to respond with just the first letter of the response. This is done by extracting the first letter of A\$ using the LEFT\$ function and then testing for a "Y" or "N". Using this method single-letter responses as well as the complete word are acceptable.

```
100 INPUT "WOULD YOU LIKE MULTIPLE CHOICE(YES/NO)";A$  
110 IF LEFT$(A$,1)="Y" THEN 200
```

Some prompts use data from previous operations in the program. An example of this is a program that generates drill and practice questions for addition. Prior to the prompt the program generates two values (A and B) which the student then adds mentally or on paper before entering the answer. This prompt might then use both the print and input statements.

```
100 PRINT "[clr dn dn dn dn dn]";
110 PRINT "WHAT IS THE SUM OF";A;" +";B;
120 INPUT S
```

Note the use of line 100 which clears the screen and moves the cursor down 5 lines before printing the prompt. This action avoids any distraction from previous questions that would otherwise remain on the screen.

A more creative solution to this problem might be to print the prompt in the form of a traditional addition question as follows:

```
100 PRINT "[clr dn dn dn dn dn]";
110 PRINT "[rt rt rt rt]";A
120 PRINT "[rt rt rt]";B
130 PRINT "[rt rt rt]_";
140 INPUT "[rt rt rt]";S
```

\_ is a shift @

If A is the value 10 and B is 15 this code displays the following:

```
10
15
—
?
```

Although this solution is a lot more work the results are far superior to previous methods and show the power of the PET/CBM in implementing an effective solution.

## DEFAULT RESPONSES

In many applications where a choice is given to the user, the response can be anticipated. A program which normally reads a file but has an option to create a file is an example of when a default would be useful. The prompt might be:

**(READ) OR (CREATE) A FILE**

Here the user must type in either READ or CREATE as a response. Since we expect that READ is the most frequent response it may become the default. This is done by printing the default value with the prompt and then moving the cursor to the left past the default value.

```
100 INPUT "(READ) OR (CREATE) A FILE READ[1t 1t 1t
1t 1t 1t]";A$
```

2 blanks

6 cursor lefts

This is what the display shows:

**(READ) OR (CREATE) A FILE? READ**

question mark created  
as a result of A\$ in  
the INPUT statement

cursor position

An example of this application is a budgeting program with a main menu that asks for annual, monthly, or daily budget items. The program then branches to a lower level menu to itemize particular entries in one of these three categories.

```

100 PRINT "[clr]ENTER A BUDGET CATEGORY"
110 PRINT "[dn rvs]A[rvs off]NNUAL"
120 PRINT "[dn rvs]M[rvs off]ONTHLY"
130 PRINT "[dn rvs]D[rvs off]AILY"
140 GET A$:IF A$="" THEN 140
150 FOR I=1 TO 3
160 IF A$=MID$("AMD",I,1) THEN ON I GOSUB
    200,400,600
170 GOTO 100
200 REM ANNUAL BUDGET ITEMS
210 PRINT "[clr]SELECT AN ANNUAL ITEM"
220 PRINT "[dn]1 - HOUSE INSURANCE"
230 PRINT "[dn]2 - CAR INSURANCE"
240 PRINT "[dn]3 - INCOME TAX"
250 PRINT "[dn]4 - PROPERTY TAX"
260 PRINT "[dn]5 - RETURN TO MAIN MENU"
270 GET A$:IF A$="" THEN 270
280 N=VAL(A$)
290 IF N=5 THEN RETURN
300 ON N GOSUB 1000,2000,3000,4000
310 GOTO 200
400 REM MONTHLY BUDGET ITEMS
    .
    .
    .
etc.

```

## FORM FILLING

Some application areas such as accounting, CAI, and testing can benefit from a technique called form filling. This method permits the program's user to input data in a predefined location on the line. Typically this location is inside a box, as in an accounting ledger, or in a blank area such as a fill-in-the-blank type of test question. The following code shows the use of form filling to enter an account number and date in a precise location within a box. In this code the lowercase "b" represents a blank character.

```

10 PRINT "[clr rvs]bbACCOUNTbbbbbbDATEbbbbbb[rvs
    off]"
20 PRINT "[rvs]b[rvs off]bbbbbbbbbb[rvs]b[rvs off]bbb
    bbbbbbbbbb[rvs]b[rvs off]"
30 PRINT "[rvs]24 spaces[rvs off]"
40 INPUT "[home dn rt]";A$
50 INPUT "[home dn 11 rt's]";D$
60 A$=LEFT$(A$,6)
70 D$=LEFT$(D$,9)

```

The above code displays a form on the screen something like this:

ACCOUNT	DATE
<input type="text"/>	<input type="text"/>

When the program asks for input the question mark appears in the box under the appropriate heading and is followed by the flashing cursor. This type of program presents a unique problem since the INPUT statement accepts all of the characters on the line following the question mark. This also includes graphic characters.

One solution is to use the GET statement in a loop but a far simpler method is to use the LEFT\$ function, which selects the number of characters desired from the input string. This of course means you need to know how many characters will be entered.

## **COMMAND LANGUAGES**

Command languages are useful for applications such as word processing, where simple prompting or the use of a menu is either impractical or too unwieldy to enter complex commands.

For instance, a typical command is to change a string from one value to another to correct a spelling error or to change a word. To change the word "error" in the previous sentence to "mistake" a command like

**C/error/mistake**

is entered.

Using a command language requires considerably more experience than simply responding to a menu, but it is much faster than using a dialogue. Programming for a command language also tends to be more complex since the program needs to recognize the type of command, often identified by the first character in the command, and the operands which can legitimately accompany that command. Often there is no prompt since this type of application requires an active operator who initiates all action.

The following code shows how the preceding command might be analyzed in BASIC.

```
100 INPUT C$
110 IF LEFT$(C$,1)="C" THEN 500
    :
    :
500 REM DECODE CHANGE COMMAND
505 S1$="": S2$="" :REM EMPTY STRINGS
510 IF MID$(C$,2,1)<>"/" THEN PRINT "COMMAND
    ERROR":GOTO 100
520 L=LEN(C$)
530 FOR I=3 TO L
540 IF MID$(C$,I,1)="/" THEN 570
550 S1$=S1$+MID$(C$,L,1)
560 GOTO 580
570 S2$=RIGHT$(C$,L-I):I=L
580 NEXT I
590 IF LEN(S2$)=0 THEN PRINT "COMMAND ERROR":
    GOTO 100
```

Statements 510 to 590 ensure that the command format is followed by checking for a slash separating the command (C) from the first string and then checking for a second string. Statements 530 to 560 extract the first string by concatenating each character to S1\$. When the end of the string is found by 540 the second string is extracted in 570 and stored in S2\$.



## **REVIEW QUESTIONS—CHAPTER 7**

1. Why is it important to consider the user level of a person who will be using our programs?
2. What are the three different levels of computer users? How do these levels relate to the types of people who will be likely to use your programs?
3. Discuss the types of prompts that may be used for interacting with a user.
4. What is meant by a default? Write an INPUT statement that asks for an INSERT or DELETE response providing the insert option as the default.
5. Define a menu. How is a menu created on the screen? Describe three possible ways of accepting the user's response to a menu.
6. Describe what is meant by a multilevel menu. Give an example of this kind of menu other than the example given in the chapter.
7. What is meant by form filling? What are the advantages and disadvantages of this type of user data entry?
8. Explain what is meant by a command language.

# 8

## ***Graphics, Animation, and Sound***

One of the particularly nice features about the PET/CBM is the availability of the graphic character set. This set consists of 62 characters directly accessible from the keyboard and four additional characters that are available using the POKE command. By using the reverse of these characters a total of 132 characters are available.

### **GRAPHIC CHARACTER SET**

Figure 8.1 shows the graphic character set with the related ASCII codes while figure 8.2 shows the graphic characters organized by type.

Most graphics are selected by pressing the appropriate shifted key and can be used in a character string like any other character. For instance, an upper left corner graphic is selected by pressing a shifted zero giving:



By using the corner symbols in keys 0 . — and = the horizontal line (shifted@) and the vertical line (shifted]) a box may be drawn with the following program.

```
10 PRINT " "
20 PRINT " "
30 PRINT " "
40 PRINT " "
```

Using the reverse key a solid box may be drawn as follows:

```
10 PRINT "rvs "
20 PRINT "rvs "
30 PRINT "rvs "
40 PRINT "rvs "
```

A solid outline with an empty interior could be produced by turning reverse on and off.

```
10 PRINT "rvs"
20 PRINT "rvs rvs off rvs"
30 PRINT "rvs rvs off rvs"
40 PRINT "rvs"
```

	99		101		120		119		113		86
	69		84		121		111		114		91
	68		71		117		116		115		78
	67		66		118		106		107		77
	64		93								
	70		72								102
	82		89		97		105		81		92
	100		103		98		95		87		104
	65		85		79		124		112		
	90		74		76		126		110		
	83		73		80		108		109		
	88		75		122		123		125		
							127				

Figure 8.1 Graphic POKE codes

Using the appropriate codes, graphics may be POKE'd to the screen instead of using the PRINT. The POKE code for a solid square (reverse space) is 160, which may be used as follows to produce a solid box.

```
10 POKE 32948,160
20 POKE 32949,160
30 POKE 32988,160
40 POKE 32989,160
```

This approach may seem more awkward than using PRINT, and for this graphic it really is. However, POKE has the advantage of executing faster than PRINT, and since the address and value may both be variables we can take advantage of this characteristic if a pattern is to be duplicated on the screen. But more on this later.

horizontal line	vertical line	wide bar	narrow bar	T shape	diagonal
					grid 
		solid half 	solid triangle 	circle 	
card suit	partial circle	corner	solid quarter	open quarter	

Figure 8.2 Graphic characters by type



```

100 REM REACTION TIMER
140 PRINT "START"
145 PRINT TAB(12)"REACTION  TIMER"
146 PRINT
150 PRINT TAB(11)" "
160 PRINT TAB(11)" | SIGNAL | TIME | "
170 PRINT TAB(11)" | "
180 PRINT TAB(11)" |  |  |  | "
182 PRINT TAB(11)" |  |  |  | "
184 PRINT TAB(11)" |  |  |  | "
190 PRINT TAB(11)" | "
195 GOSUB 700
200 GOSUB 500
210 GOSUB 540
220 INPUT "GO! TRY AGAIN (Y/N)";A$
235 IF LEFT$(A$,1)="Y" THEN 100
240 STOP
500 REM DISPLAY READY SIGNAL
510 GOSUB 600
520 PRINTTAB(13)"READY";
530 GOSUB 700
535 RETURN
540 REM DISPLAY GO SIGNAL
542 GOSUB 600
544 PRINTTAB(13)"  GO  " ;
546 OT=TI
548 GET A$:IF A$="" THEN 548
550 NT=TI
551 ET=INT((NT-OT)/60*1000)/1000
552 PRINTSPC(3);ET
554 RETURN
600 PRINT "XXXXXXXXXXXXX";
610 RETURN
700 REM TIME DELAY
710 FOR I=1 TO 1500:NEXT I
720 RETURN

```

**Figure 8.5** Reaction Timer program

Figure 8.5 contains the program for the timer. Once the scoreboard has been drawn the word 'READY' is displayed in the box under the title START. This action requires cursor positioning with subroutine 600 and a TAB(13) in line 520. Using both cursor control and TAB permits the program to display a value (READY) inside the box without destroying the lines that are currently on the screen.

A time delay is then used before the 'GO' signal is displayed. Notice that the string containing GO in line 544 contains extra spaces to ensure all the letters of START are cleared from the box.

The remainder of the program is basically like before except for line 551. This statement calculates the elapsed time (ET) and reduces the answer to three decimal places so it will fit in the box.

## USING POKE TO PRODUCE A GRAPHIC

The introduction to this chapter mentioned that both PRINT and POKE could be used to produce graphics. POKE generally operates faster than PRINT and also permits the use of variables, which will be valuable for the chessboard graphic. Speed is primarily of interest for animation.

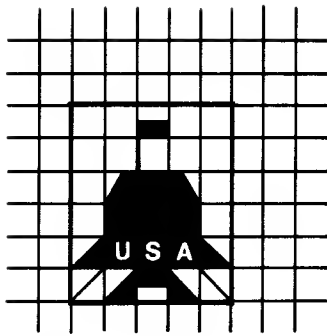


Figure 8.6 Lunar lander working drawing

### Lunar Lander

An all-time graphic favorite is the lunar lander, and we will produce it here using POKE statements. First a graph paper drawing of the LEM is needed to work from. This is given in figure 8.6.

The next step is to determine the ASCII POKE codes for each of the characters in the graphic. Using figure 8.1 select each code (remembering that some of them are reverse graphic characters) and write them down in the order that they appear in the design. Now you will have a chart as follows:

32	32	121	32	32
32	32	101	32	32
32	233	160	223	32
32	160	160	160	32
233	149	147	129	223
78	233	120	223	77

Now we have a choice. Either a separate POKE can be used for each character, which requires 30 POKEs, or we could read these numbers as data and POKE them using a variable. This is the preferred solution as shown in figure 8.7. Notice the nested FOR loops that control the reading and POKE'ing. The starting value (33064) of the outer value controls the screen addresses which are POKE'd with the data.

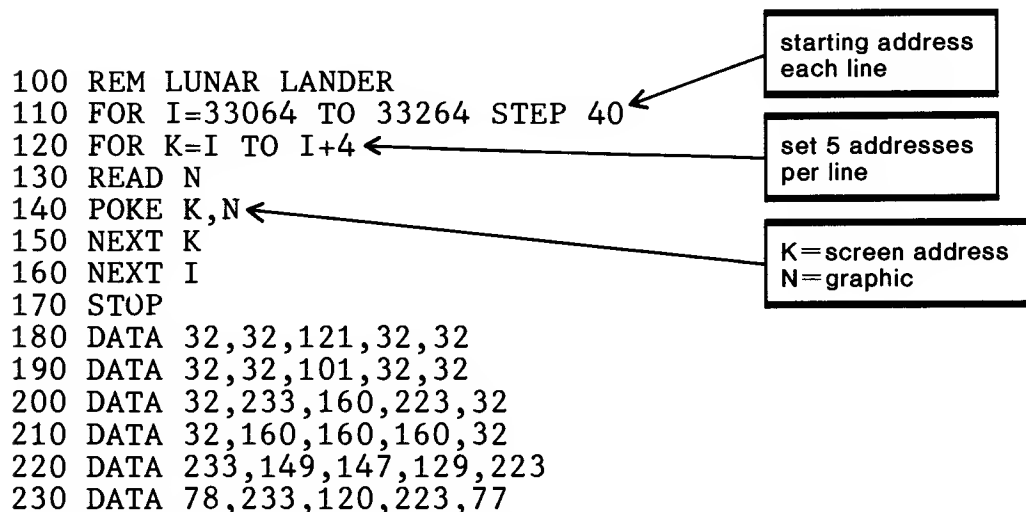


Figure 8.7 Lunar lander using POKEs (for PET and CBM 4000 series only)

```

80 PRINT "[clr]
90 N=32768
94 FOR S=1TO8
95 FOR R=1TO3
100 FOR P=1TO32STEP8
110 FOR P1=1TO4
120 POKE N,160
125 N=N+1
130 NEXT P1
135 N=N+4
140 NEXT P
145 N=N+8
150 NEXT R
160 IF (INT(S/2)*2) <> S THEN N=N+4:GOTO 170
165 N=N-4
170 NEXT

```

**Figure 8.8** Chessboard graphic program (for PET and CBM 4000 series only)

### **Chessboard**

The problem of displaying a chessboard seems at first quite simple. However this simplicity is soon overshadowed by two problems. The first occurs when we consider the number of POKE statements needed. If the board is to consist of  $3 \times 3$  squares on the screen then we need  $3 \times 3 \times 8 \times 8$  (a board has 8 squares to a side) POKEs. Obviously 576 POKEs are too many. Since each square is either black (blank screen) or white (ASCII 160) then possibly we can compute the value to be POKE'd and use variables.

A second problem may not be apparent until our first attempt to display the chessboard. If each square on the board is indeed  $3 \times 3$  then a square that is longer than it is wide will display. The reason for this surprise is that each character on the screen has about a 4:3 height to width ratio. This problem may be corrected by making each square on the board 4 characters wide and 3 characters high. The program is in figure 8.8

### **ANIMATION**

Animation is the process of taking a graphic and causing it to move up, down, left, right, or diagonally on the screen. Usually animating an object on a microcomputer is done using cursor controls in a PRINT statement or by recalculation of a series of POKE addresses. Normally, the Hollywood cartoon approach of creating many frames of an object and displaying them rapidly is not used. The reason to avoid this approach, if possible, is that it requires an excessive amount of storage and a substantial amount of time to type in all the images.

#### **Rocket 1**

Figure 8.9 is a program to create a rocket blasting off. This is one of the easiest animation techniques to master. The program begins with a clear screen and positions the cursor 11 lines from the bottom of the screen by printing the variable A\$ which contains cursor down characters.

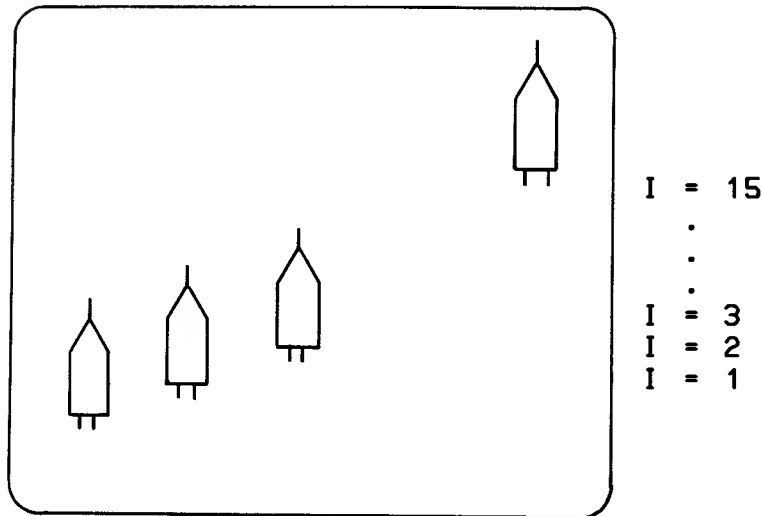
```

60 A$="dn dn dn dn dn dn dn dn dn dn dn dn dn dn"
95 PRINT LEFT$(A$,15-1);

```



At this position the rocket is displayed using PRINT statements. Next the FOR loop causes the cursor to be positioned one line higher than for the previous output by selecting 15-I characters from A\$. Now when the rocket prints it is one line higher on the screen than before. Since this happens relatively quickly, motion is produced and the rocket takes off.



One other important feature of the program is the last PRINT statement, which simply prints a blank line of characters. This is necessary to wipe out the previous bottom line of the rocket, which will otherwise stay on the screen as each new rocket image is displayed.

```
60 A$=" "
70 PRINT " "
80 FOR I=1 TO 15
85 PRINT " "
95 PRINT LEFT$(A$,15-I);
100 PRINT " | "
110 PRINT "  " "
130 PRINT "  " "
140 PRINT "  " "
150 PRINT "  " "
160 PRINT "  " "
170 PRINT "  " "
175 PRINT "  " "
180 NEXT
```

**Figure 8.9** Rocket 1 program

## Rocket 2

Animation can also be produced by using cursor controls directly in the print string. This is a technique used extensively in the rather amusing program Toker. Although not as easy to read, the program in figure 8.10 has the advantage of being quite short. A second benefit will be seen when the program is run. Because of the cursor movement the motion of the rocket is very smooth since multiple PRINTs are unnecessary. The only problem with this method is the difficulty in controlling the speed of the movement, which could be accomplished with a delay loop in the previous program.





## Rolling Die

The rolling of dice presents an interesting problem in animation since it requires the changing of the die image as it rolls and also the random generation of from one to six dots at the end of the rolling sequence. To simplify the program somewhat the die will be seen in only two dimensions and will present only two images (figure 8.12) as it rolls.

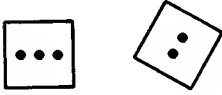


Figure 8.12 Two images for the rolling die

To display these images a series of POKEs are used. The addresses POKE'd are adjusted using a FOR loop to move the alternating images across the screen. You may ask why the dots on the die surface are not varied randomly as the roll takes place, as they would be with real dice. The reason for not doing this during the roll is that the time needed to generate the dots is fairly long. A second, equally important reason is that the motion is fast enough that the dots appear to be changing, since the program alternates between three dots and two dots during the roll.

The patterns used for POKE'ing the images are in figure 8.13.

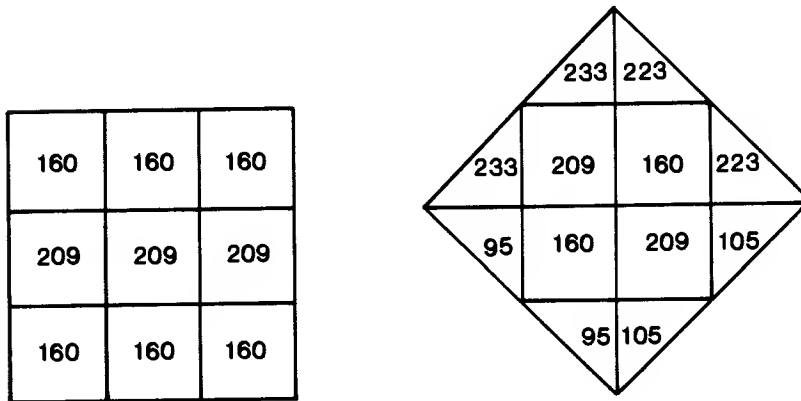
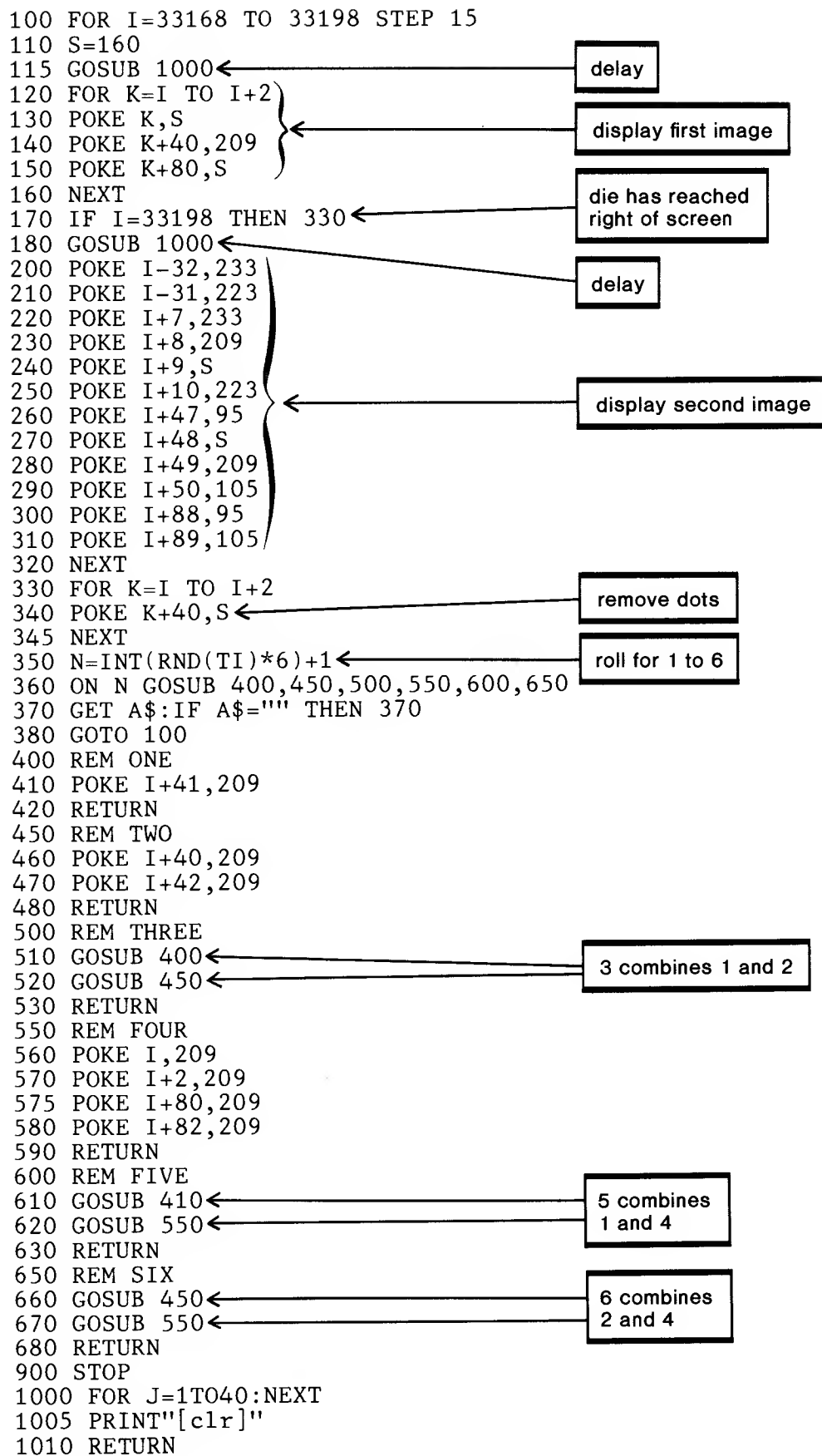


Figure 8.13 POKE values for die images

These images are POKE'd in an alternating pattern across the screen with statements 100 to 320.

When the die comes to rest at the right side of the screen the three dots are removed in statements 330–345 in preparation for the random generation of dots (figure 8.14). A value from 1 to 6 is generated in 350 and stored in N. The ON GOSUB in 360 then selects the appropriate subroutine to generate from one to six dots. These subroutines have been simplified by combining operations when possible. For example, a 3 consists of a combination of 1 and 2 so if three dots are to be generated, subroutine 3 simply calls subroutines 1 and 2. A 5 consists of 1 and 4 and a 6 consists of 2 and 4.



**Figure 8.14** Program to roll a die (for PET and CBM 4000 series only)

Memory Address	POKE Value	Explanation
59464	0 to 255	—Frequency Control
59466	0 15	—Shift Register Off Generate Sound
59467	0 16	—Control Register Normal Setting for Tape Generate Sound

**Figure 8.15** Addresses for producing sound

## GENERATING SOUNDS

Both the PET and CBM computers are equipped with the ability to produce sounds. Early models required a separate speaker/amplifier to be attached to the CB2 output on the Parallel User Port. More recent models have a built-in speaker and thus do not require a separate attachment.

In general, sound is produced by POKE'ing memory addresses to produce the desired sound effects. The addresses used are listed in the chart shown in figure 8.15. These addresses are valid for all PET and CBM computers.

The basic approach for producing sound is to set the shift register and the control register to 15 and 16 respectively using POKE commands in the following order:

```
10 POKE 59467,16
20 POKE 59466,15
```

Now the computer is ready for the command to produce a tone. This requires address 59464 to be POKE'd with a value to set the tone's frequency. This value may be anything from 0 to 255 with 0 being the highest tone and 255 the lowest. Each value between this range will produce a slightly different frequency.

To stop the tone addresses 59466 and 59467 are POKE'd back to zero. Now try this code:

```
10 POKE 59467,16 }
20 POKE 59466,15 }
30 POKE 59464,150 }
40 POKE 59467,0 }
50 POKE 59466,0 }
```

When you RUN this short program you will get a short beep. You might even miss hearing it if you're not alert. The reason for the short duration is that you are hearing the tone for the length of time that statement 30 requires for execution on the computer, which is just a fraction of a second.

To make this tone last longer we need a delay loop between statements 30 and 40 as follows:

```
10 POKE 59467,16
20 POKE 59466,15
30 POKE 59464,150
35 FOR I=1 TO 500:NEXT
40 POKE 59467,0
50 POKE 59466,0
```

Now we get tone for about a second. The larger the value in the delay loop the longer the tone will last.

Now let's try placing the POKE inside the loop and using the loop variable for the POKE value. In this case we must be careful not to go outside of the range of 0 to 255 since we are using this as a POKE value.

```

10 POKE 59467,16
20 POKE 59466,15
30 FOR I=1 TO 120
40 POKE 59464,I
50 NEXT I
60 POKE 59467,0
70 POKE 59466,0

```

I produces a varying tone

This simple code will produce a tone which starts at a high note and slides down to a much lower one. The reverse of this with a tone beginning at a low one and sliding up to a high note can be produced by changing the FOR loop to the following:

```

30 FOR I=120 TO 1 STEP -1

```

Experiment with this FOR loop by trying different ranges of values and even different STEP values.

Nested FOR loops can be used to control not only the frequency of the tone but also the number of times the tone is repeated. Try enclosing the previous FOR loop inside a second loop for 3 repetitions of the tone sequence.

```

10 POKE 59467,16
20 POKE 59466,15
25 FOR K=1 TO 3
30 FOR I=1 TO 120
40 POKE 59464,I
50 NEXT I
55 NEXT K
60 POKE 59467,0
70 POKE 59466,0

```

I-loop

vary tone from high to low

K-loop

repeat sound 3 times in succession

Now try this program. What do you think it sounds like? Observe how two loops are used; one to go up the frequency scale and the second to go down.

```

10 POKE 59467,16
20 POKE 59466,15
30 FOR K=1 TO 25
40 FOR I=240 TO 10 STEP -8
50 POKE 59464,I
60 NEXT I
70 FOR I=10 TO 240 STEP 8
80 POKE 59464,I
90 NEXT I
100 NEXT K
110 POKE 59467,0
110 POKE 59466,0

```

## MUSIC PLAYER

The examples of sound used so far have been sliding tones that might be used to develop different sound effects from the PET/CBM. But not all sounds need be of this type. Music is generally produced with distinct tones, each with precise duration. This is quite readily achieved using the POKE for sound as we have already discussed.

The basic procedure, as before, is to POKE the control and shift registers, POKE the frequency, and then go into a wait loop for the duration you want to hold the note. Figure 8.16 shows the values of the frequency for different notes on the musical scale. For instance, if you wanted a low C then you would POKE 59464 with the value 255. A middle C would be 126.

Note		POKE Value
Low	C	255
	D	226
	E	200
	F	187
	G	170
	A	150
Middle	B	133
	C	126
	D	110
	E	99
	F	93
	G	83
High	A	75
	B	65
	C	62
	D	55
	E	48

Figure 8.16 POKE values for music

The next program, in figure 8.17, uses these values to make a music player out of the PET. The keyboard is set up as a musical instrument, a piano or organ if you like, with each note associated with a key as follows:

Key	A	S	D	F	G	H	J	K	L	:	Z	X	C	V	B	N	M
Note	C	D	E	F	G	A	B	C	D	E	F	G	A	B	C	D	E

As you can see each note follows consecutively across the bottom two rows on the keyboard. The keys were selected here for the PET. The CBM should substitute the (;) for the (:) to get a correct sequence.

The program reads the frequency values from a DATA statement into array C. The keyboard letters which relate to these values are in variable N\$. When a key is pressed, statements 210 to 230 search for the key in N\$ and then select the corresponding note from the array. For instance, if L is pressed (representing the note D) a match is found in position 9 of N\$. Element 9 of the array (value 110) is then used as a POKE value in statement 260 and a D note is played.



```

100 REM MUSIC PLAYER
110 PRINT "[clr]"
120 DIM C(17)
130 N$="ASDFGHJKL:ZXCVBNM"
140 GOSUB300
150 FOR I=1 TO 17
160 READ C(I)
170 NEXT I
180 DATA 255,226,200,187,170,150,133,126,110,99
190 DATA 93,83,75,65,62,55,48
200 GETA$:IF A$=""THEN POKE 59467,0:GOTO 200
210 FOR J=1 TO 17
220 IF A$=MID$(N$,J,1) THEN 250
230 NEXT J
240 GOTO 200
250 POKE 59467,16
260 POKE 59464,C(J)
270 POKE 59466,15
280 FOR I=1 TO (6-S)*100:NEXT I
290 GOTO 200
300 PRINT:PRINT
310 PRINT "ENTER SPEED FROM 1 TO 5";
320 INPUT S
330 IF S>5 THEN 300
340 PRINT
350 PRINT "FOR NOTES USE KEYS ";N$
360 PRINT
370 PRINT "PRESS STOP TO QUIT"
380 RETURN
390 END

```

*Figure 8.17* Music player

## **REVIEW QUESTIONS—CHAPTER 8**

1. Write some print statements to produce a graphic triangle on the screen.
2. Produce the same triangle as in question 1 using POKE statements.
3. Draw a graphic with changing values to simulate a display of a digital watch.
4. Try to change the program above to include stop-watch capabilities.
5. Using POKES write a program to fly an airplane across the screen.
6. Can you make the above airplane drop a bomb when in flight?
7. What addresses need to be POKE'd to produce sounds on the PET/CBM? How are these addresses used to produce a steady tone?
8. How would you produce a beep beep sound?
9. Write a program that will play a short tune when it is RUN.

# 9

## ***Tape Files***

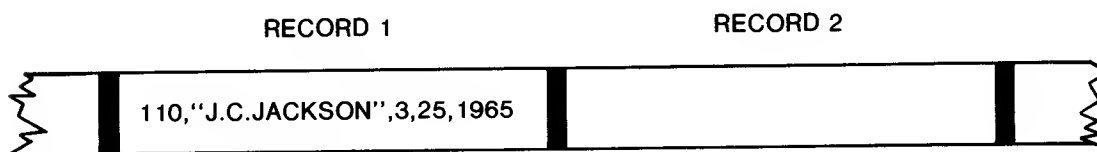
### ***Extend Your Reach***

**W**hat kind of record keeping would you like to do on your PET or CBM? Personal or business accounts? Financial records? Student records? Or maybe you are a collector of books, records, stamps, or even baseball cards. The use of files in the PET or CBM allows you to store data on tape or disk files that are external to the program. In addition to eliminating or reducing the need for DATA statements, files permit the data to be accessed by any other BASIC program. This is not the case with the DATA statement, which may only be accessed by the program containing it.

Files may be stored and accessed in several ways depending upon whether tape or disk is used. Sequential files are used on either tape or disk and are the easiest technique to master. Direct access or relative files are only available with disk but represent an important technique to be used for handling large sets of data. In this chapter we will restrict our discussion to sequential tape files.

#### **CONCEPTS**

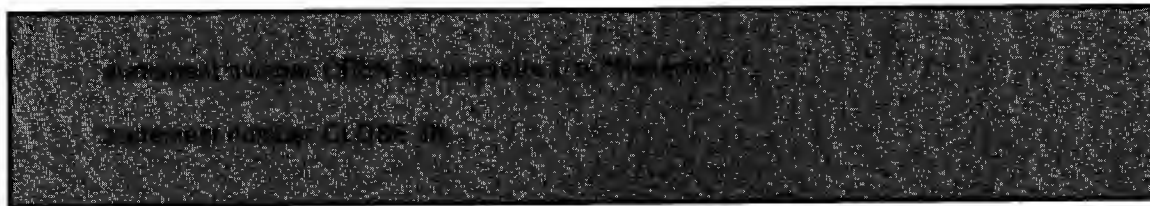
Sequential files may be thought of as DATA statements that reside on magnetic tape. Data on tape files are read or written sequentially, starting at the beginning of the file and progressing through each item of data until the end of file is reached. Figure 9.1 shows how data appears on a tape file. The major differences between sequential files and DATA statements are the lack of a statement number and the missing keyword DATA. In other words just the data are present on the file.



**Figure 9.1** A sequential tape file

An important characteristic of tape is its ability to store records sequentially. Since these records do not form part of the computer's memory, except as they are read by the program and depending upon the type of application, files may sometimes exceed the size of the memory capacity.

## OPEN, CLOSE



Before the program can write data on the tape or read from an existing tape file it is necessary to open the file. Opening makes the file available to our program, defines whether it will be input or output, and gives it a number and a name. The OPEN statement is used for this purpose and has the following format:

```
10 OPEN a,b,c,"name"
```

where: a is an integer from 1 to 255 used to identify the file.  
b is 1 for cassette 1 and 2 for cassette 2. Normally only cassette "one" is used.  
c specifies read or write as follows:  
0—tape is input  
1—tape is output with an End-of-File (EOF) marker when it is closed.  
2—tape is output with an End-of-Tape (EOT) marker when it is closed.  
name—is the name used to identify the file.

Let's try a simple example of how this OPEN statement works. If you wanted to create a tape file with data representing your budget the following OPEN could be used:

```
100 OPEN 1,1,1,"BUDGET"
```

file number      cassette unit      EOF marker      filename

This statement will open file number 1, on cassette number 1, to be written with an EOF marker following the last record on the file. The file on the tape will be identified with the filename BUDGET. When this file is read later the program can then check the filename, which is recorded at the beginning of the file, to ensure that we are reading the correct file.

After the file has been created it must be closed with the CLOSE statement. Closing writes the EOF or EOT marker on output files and also releases the tape for other operations by the program. For example, an output file could be closed and then opened for input in the same program.

The CLOSE statement simply identifies the file number:

```
200 CLOSE 1
```

A program that reads and processes a tape file will have the following basic structure:

```
100 OPEN 1,1,0,"BUDGET"  
:  
:  
↑  
[statements to read and process the file]  
↓  
:  
:  
200 CLOSE 1
```

Notice in this example that the third parameter in the OPEN is a zero, which identifies the file as output.

### **PRINT#**



Data are placed on the tape(output) by the PRINT# statement. The number used after the # symbol is the file number assigned to the tape in the OPEN statement. In this case a 1 has been used.

```
100 PRINT#1,A5
```

This statement indicates that the contents of variable A5 is to be written (printed) on tape number 1.

### **WRITING BUDGET NAMES ON TAPE**

Figure 9.2 contains a program that accepts names of budget items from the keyboard and writes them sequentially on the tape.

```
100 REM CREATE NEW FILE
110 PRINT CHR$(147):REM CLEAR SCREEN
120 OPEN 1,1,1,"BUDGET"
130 INPUT "ENTER ITEM NAME (END)";N$
140 IF N$="END" THEN 170
150 PRINT#1,N$
160 GOTO 130
170 CLOSE 1
```

**Figure 9.2** PRINT Budget item names on tape

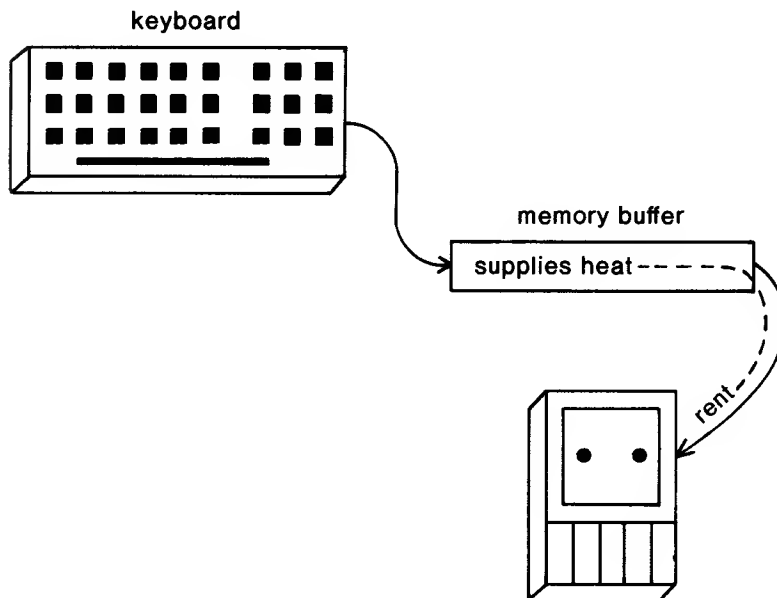
The program begins with the message:

**PRESS PLAY AND RECORD ON TAPE #1**

The program then asks the user for an item name or the word END to indicate all of the data has been entered. The following is how the screen will appear after a number of entries have been made from the keyboard.

```
ENTER ITEM NAME (END)? RENT
ENTER ITEM NAME (END)? TELEPHONE
ENTER ITEM NAME (END)? LIGHT
ENTER ITEM NAME (END)? HEAT
ENTER ITEM NAME (END)? SUPPLIES
ENTER ITEM NAME (END)? END
```

As these names are being entered on the keyboard no activity will be apparent on the tape. Eventually, if enough items are entered or END is reached the tape will then write. What is happening here is that the data is first being stored in a memory buffer in the PET. This buffer can hold a maximum of 191 characters; so only when it has been filled with data or the file is closed will the PET then write the data from the buffer to the tape drive. The internal use of a buffer does not usually affect the way you will write tape programs.



The budget item names will appear as follows on the tape:

```

RENT(cr)TELEPHONE(cr)LIGHT(cr)HEAT(cr)SUPPLIES(cr)

```

Since each item is being printed individually by statement 150 a carriage return (cr) will follow each item. The carriage return is an ASCII code of 13 and is necessary to separate each item of data. In this example, the carriage return is included automatically since we are printing only one item at a time to the tape.

### **INPUT#**

```

statement number INPUT# the list of variables

```

Data is read from the tape in a manner similar to the keyboard. But instead of using INPUT the INPUT# statement is used. The statement,

```
100 INPUT#1,NA
```

reads a numeric value from the tape identified as 1 in the OPEN and places the value in the variable NA. Now the need for the carriage return character on the tape is evident. When the number for NA

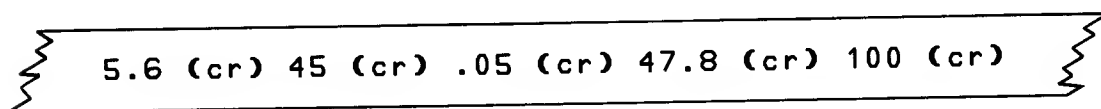
is read the carriage return signals the end of this item, just like pressing Return on the keyboard after keying a number.

When data is read from tape it is necessary to know when all of the data has been read. In other words we need to know when end of file is reached. There are two ways to do this. The first method requires that you know exactly the number of items to be read. The second uses the PET's status (ST) code, which signals the program when end of file is reached.

To use the first method, suppose we want to read and display 5 numbers from a file called NUMBERS. The following program could be used:

```
100 OPEN 1,1,0,"NUMBERS"
110 FOR I=1 TO 5
120 INPUT#1,NA
130 PRINT NA
140 NEXT I
150 CLOSE 1
```

This program reads a tape with 5 numbers on it separated by carriage returns as follows:



5.6 (cr) 45 (cr) .05 (cr) 47.8 (cr) 100 (cr)

### **READING THE BUDGET NAMES TAPE**

Figure 9.3 contains a program to read the budget item names created on tape in figure 9.2. This program opens the BUDGET file as input, reads each name, and displays it on the screen. Since this program does not use a fixed number of data items the ST code is used to detect end of file.

A status of 64 is a signal from the operating system that end of file has been reached. The ST code is tested in statement 260 prior to the input of a new data item. When the code is equal to 64 then the last item read was the final item on the tape and the program branches to 300, where the file is closed.

A new approach to clearing the screen is also used in this program. Line 230 prints the CHR\$ value of 147 which, as Appendix F shows, is the character code equivalent to the CLR key. This approach to using the clear or other functions is useful when a printer that does not print the graphics characters is used with your PET or CBM.

```
175 REM READ BUDGET NAMES
180 PRINT "REWIND TAPE NO. 1"
190 PRINT "PRESS ANY KEY TO CONTINUE"
200 GET A$:IF A$="" THEN 200
210 REM READ AND DISPLAY FILE
220 OPEN 1,1,0,"BUDGET"
230 PRINT CHR$(147)
240 PRINT "BUDGET ITEM"
250 PRINT
260 IF ST=64 THEN 300
270 INPUT#1,N$
280 PRINT N$
290 GOTO 260
300 CLOSE 1
```

**Figure 9.3** Program to read budget item names

The screen display from this program is as follows:

```
REWIND TAPE NO. 1
PRESS ANY KEY TO CONTINUE
PRESS PLAY ON TAPE #1
```

The screen then clears and displays the following:

```
BUDGET ITEM

RENT
TELEPHONE
LIGHT
HEAT
SUPPLIES
```

### **HOW TO HANDLE RECORDS WITH MULTIPLE FIELDS**

Most data processing files require the use of more than one field per record. In a payroll program fields such as Employee Number, Rate, Hours, Tax, and so forth are necessary for complete information. As a result each Input or Print operation on tape requires a variable for each of these fields.

We have already seen how each value on tape is followed by a carriage return character. This character separates the data from a following data item. When multiple fields are used each field in the record must also be separated by a carriage return. Figure 9.4 shows a program to create a budget file that contains both an item name and an amount.

```
100 REM CREATE NEW FILE WITH MULTIPLE FIELDS
110 PRINT CHR$(147): REM CLEAR SCREEN
120 OPEN 1,1,1,"BUDGET"
130 INPUT "ENTER ITEM NAME (END)";N$
140 IF N$="END" THEN 180
150 INPUT "ENTER AMOUNT";A
160 PRINT#1,N$;CHR$(13);A
170 GOTO 130
180 CLOSE 1
```

**Figure 9.4** Create budget file with name and amount

The carriage return character is included after the name by using the CHR\$ function with the ASCII value 13.

```
160 PRINT#1,N$;CHR$(13);A
```

Notice that the fields are separated by semicolons and not commas. Using semicolons keeps the fields tightly packed and saves space on tape.

Another way to create this file would be to use two PRINT# statements:

```
160 PRINT#1,N$
165 PRINT#1,A
```

It is also possible to insert commas between fields as separators:

```
160 PRINT#1,N$;",",A
```

```

100 REM READ AND DISPLAY FILE
110 PRINT "REWIND TAPE NO. 1"
120 PRINT "PRESS ANY KEY TO CONTINUE"
130 GET A$:IF A$="" THEN 130
140 OPEN 1,1,0,"BUDGET"
150 PRINT CHR$(147)
160 PRINT "ITEM","AMOUNT"
170 PRINT
180 IF ST=64 THEN 220
190 INPUT#1,N$,A
200 PRINT N$,A
210 GOTO 180
220 CLOSE 1

```

**Figure 9.5** Program to read multiple fields

Figure 9.5 shows how to read this file. Since the field separation has been implemented in the create program, using the file as input is now quite natural. As before ST is used to detect end of file.

### **UPDATING A TAPE FILE**

Most files store data that change frequently, which is where files are superior to using DATA statements. Instead of changing each program that uses the data, only the data file itself needs to be changed. These changes are then reflected in each program that accesses the data.

For this exercise we will only consider how to change existing items in the file and not how to add or delete data. As a further limitation we will limit the file to a maximum of 20 items. Obviously these are serious limitations but for now this approach will help us to understand the concepts.

The program will read the Budget file into two arrays. The first, N\$, will contain the names, and the second, A, will hold the dollar amounts. After updating has been done to the necessary items in the array the file will be rewritten from the array onto tape. Figure 9.6 contains the flowchart for this program and figure 9.7 contains the program.

### **SUMMARY OF TAPE FILES**

Tape is a slow but inexpensive means for storing data outside of your program. As we have seen from the programs in this chapter, tape can only be read sequentially. To get to any record in the file the program must read all of the records prior to the one we really want. For some applications this can be a time-consuming wait that leads to some frustration on the part of the user of the program.

This problem may be partially minimized by reading all of the file into an array before the program begins. Although this will take time initially, no further reading of the file will be necessary by the program no matter how long it is to be running. Of course, this approach means there must be sufficient space in memory to store all of the data, which is not always possible.

The last program in this chapter showed the basic method for updating a file by reading it into memory, making changes to it, and then rewriting it back onto the tape. Again, this approach requires that there be sufficient space in memory to store the entire file. If there isn't enough room one solution might be to divide the file into two or more separate files. But be careful about overwriting one file on top of another.

Another solution, on the PET, is to use two tape units. This gets a bit complicated but basically it involves reading records from one file, changing the record contents if required and then immediately writing the record onto the second tape. With this approach only one record is retained in memory at one time, rather than the entire file. Although this procedure will work it is slow! If this becomes a common need maybe this is the time to consider moving to disk.



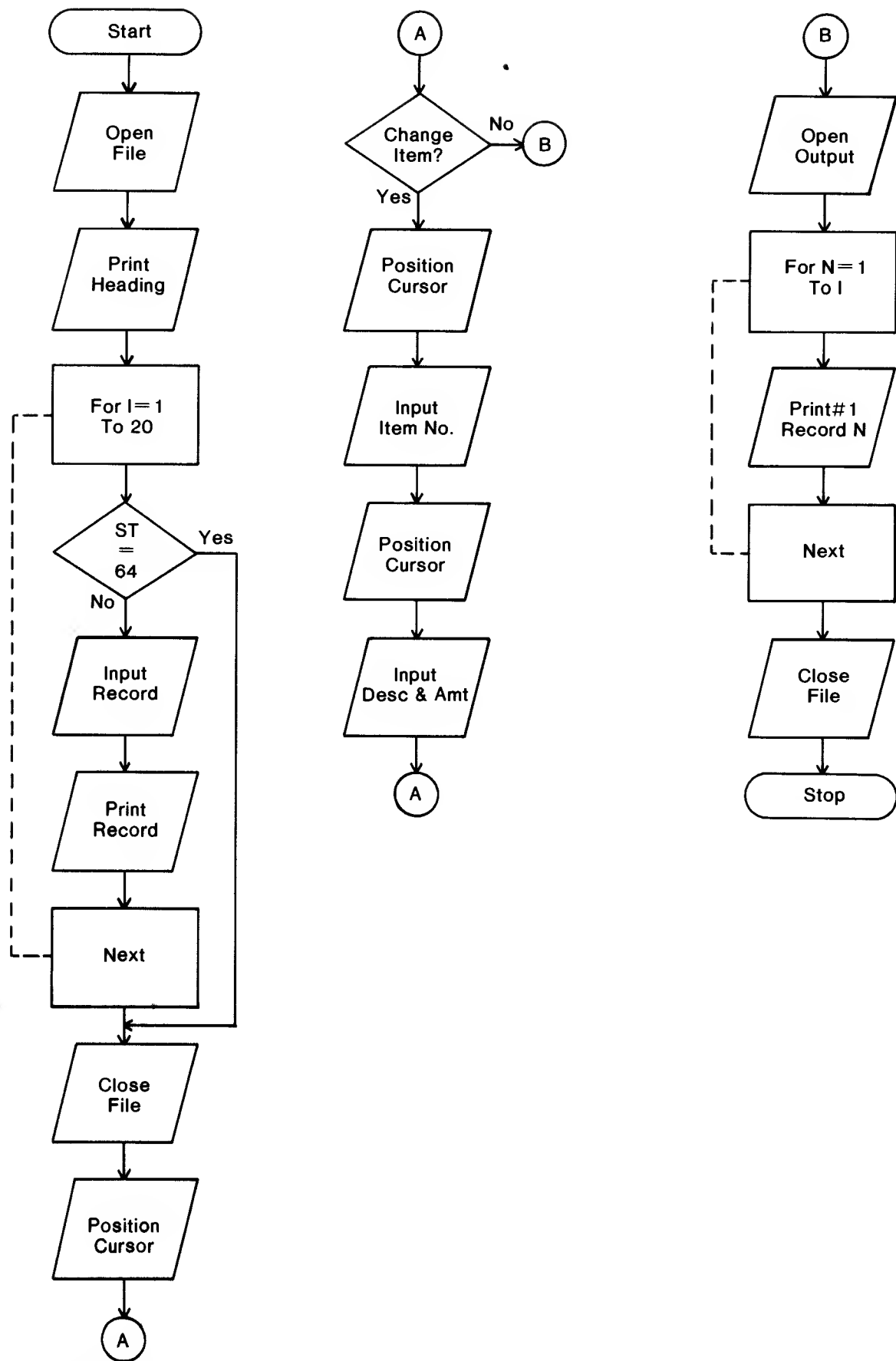


Figure 9.6 Updating a tape file

```

100 REM UPDATE BUDGET FILE
105 DIM N$(20),A(20)
110 PRINT CHR$(147);
120 OPEN 1,1,0,"BUDGET"
130 PRINT "[clr]NUMBER","ITEM","AMOUNT"
140 PRINT
150 FOR I=1 TO 20
160 IF ST=64 THEN 200
170 INPUT#1,N$(I),A(I)
180 PRINT I,N$(I),A(I)
190 NEXT
200 CLOSE 1
210 I=I-1 :REM ADJUST FOR EOF
220 REM ACCEPT UPDATES
230 GOSUB 500
240 PRINT "[rvs]DO YOU WISH TO CHANGE AN ITEM(Y/N)[off]";
250 GET A$:IF A$="" THEN 250
260 IF A$="N" THEN 340
270 IF A$<>"Y" THEN 230
280 GOSUB 500
290 INPUT "[rvs]ITEM NUMBER[off]";N
300 GOSUB 500
310 PRINT "ENTER (DESCRIPTION,AMOUNT)"
320 INPUT N$(N),A(N)
330 GOTO 230
340 REM REWRITE UPDATED FILE
350 GOSUB 500
360 PRINT "REWIND TAPE #1"
370 PRINT "PRESS ANY KEY TO CONTINUE"
380 GET A$:IF A$="" THEN 380
390 OPEN 1,1,1,"BUDGET"
400 FOR N=1 TO I
410 PRINT#1,N$(N);CHR$(13);A(N)
420 NEXT N
430 CLOSE 1
440 STOP
500 REM POSITION CURSOR
510 D$=" [22 down's]
520 C$="
530 PRINT "[home]";D$;C$
540 PRINT C$;"[2 up's]"
550 RETURN

```

maximum of 20 items

display each item read

select item to change

revise

rewrite update file

**Figure 9.7** Program to update the budget file

## **REVIEW QUESTIONS—CHAPTER 9**

1. Describe how data appears on a tape file.
2. Tape files are called sequential files. What is the advantage of a sequential file? What are the disadvantages?
3. Why is it necessary to use **OPEN** with tape files? What information does the **OPEN** specify?
4. What must be done when a program has finished using a tape file?
5. When data is being **PRINTed** to tape why is the tape not moving as each item of data is entered?
6. What commands are used to write data on tape and to read from tape?
7. You are writing a program to record Subscription Number (S), Name (N\$), and Termination Date (T\$) for each subscriber to a publication. Write a single print statement to create a tape record with these three fields.
8. Write a program that permits changes to the termination date on the file in question 7.

# 10

## Disk Files

If tape files extend your reach beyond the capacity of the computer's main memory then this is even more true of the floppy disk. The 4040 floppy disk drive can store about 170,000 bytes of data on one 5 1/4" floppy disk. Since there are two drives, more than 340,000 bytes of data are available at one time.

CBM owners can use the 4040 or the 8050 disk with about one million bytes of capacity on two drives. Then there is the hard disk with even greater capacity. In this chapter our discussion will be limited to the floppy disk units.

Figure 10.1 shows how data or programs are written on the disk. Each floppy disk surface is divided into a number of tracks and each track into a number of sectors. The exact number of tracks and sectors is not terribly important to our discussion but for the curious there are 35 tracks and a total of 690 sectors on a 4040 disk. A sector can store 256 bytes. In any case the DOS (Disk Operating System) program in the disk drive takes care of all this technical stuff.

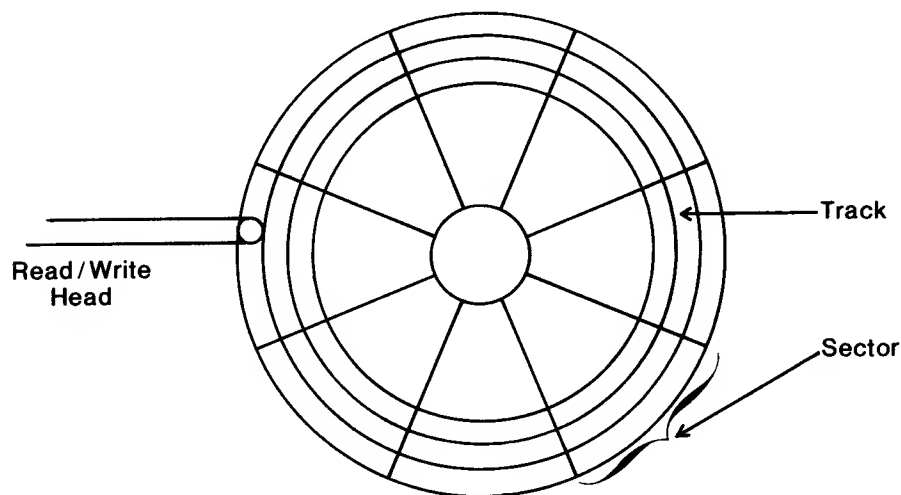


Figure 10.1 Floppy disk

When the floppy disk is inserted in the disk drive the read/write head comes in contact with the surface of the disk. Under control of the program, data is written to the surface of the disk or read from the disk into the program. In addition to data files, programs may also be stored on the disk.

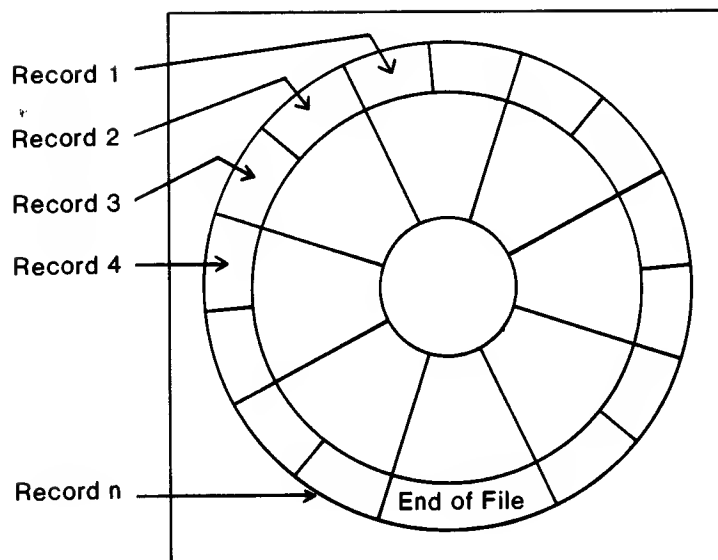


Figure 10.2 A sequential file on floppy disk

## SEQUENTIAL FILES

Using a sequential file on disk is not much different from on tape. Although the physical characteristics of disk are quite different from tape, records are logically organized in the same way. Each record on a sequential file logically follows the preceding record as shown in figure 10.2. As for tape a buffer in memory is filled with data before it is written to the disk.

When we were using tape records we discovered that to get to any record all of the preceding records needed to be read. This is also true of sequential disk files. In fact all of the limitations for sequential tape apply to sequential disk. Well, all that is except for one: Speed. Disk, as you will find, is considerably faster than using a tape file.

## DOPEN and DCLOSE

Disk uses its own open and close statements, which simplify the use of disk files. These commands were first available on PETs or CBMs with BASIC 4.0. If yours is an earlier machine you should refer to the Commodore manual for the format of the Open and Close statements.

The DOPEN has a number of options for both sequential and relative files. To keep things simple only the necessary items for sequential files will be shown now. Later additional options will be discussed for use with relative files.

DOPEN#fn, "filename", Dn, [W  
R]

where fn refers to the file number you will use in the program to reference this file. Normally a number less than 128 will be used. Most programmers use a single digit number for a filename.

"filename" is the name that will be used to identify this file. When a file is to be read it will be opened only if the filename matches.

Dn D refers to Drive and n to the drive number. If the disk is on drive 1 then the entry is D1. If drive 0 then use D0 or simply leave this entry out and it will default to drive 0.

W the W refers to an output file which will be Written. R refers to an input file that will be Read. If no entry is made then R is assumed.

A few examples are in order now to explain how the DOPEN may be used.

```
200 DOPEN#1,"ACCOUNTS",D1,W
```

          ↑          ↑          ↑          ↑  
      filenumber  filename  drive  write

In this case a file named ACCOUNTS is opened as filenumber 1. The floppy disk will be in drive 1 and is to be written on as output.

```
150 DOPEN#5,"CLASS",W
```

A file called CLASS is opened as number 5. It will be written on as an output file, and by default is on drive 0.

```
300 DOPEN#8,"RECORDS"
```

This file is number 8, and is called RECORDS. The disk defaults to drive 0 and the file will be read as input, also by default since R is not specified.

By comparison the DCLOSE statement is quite simple as shown in the following general format:

```
DCLOSE#fn
```

where

fn is the filenumber to be closed.

For example, the statement:

```
500 DCLOSE#1
```

will close filenumber 1. No additional entries are normally needed in the DCLOSE statement.

### **PRINT# and INPUT#**

Once the disk file has been opened it can be referenced using the PRINT# and INPUT# statements in the same way as for tape files. The filenumber in both the PRINT# and INPUT# refers to the equivalent filenumber used in the DOPEN statement. Here are some sample statements:

100 PRINT#5,A\$	writes variable A\$ on file 5
200 PRINT#3,K	writes numeric value K on file 3
300 INPUT#1,X\$	reads string X\$ from file 1
400 INPUT#2,S	inputs variable S from file 2

### **Writing Budget Names on Disk**

In the tape chapter we wrote a few programs that created and read a sequential tape file for budget data. Now let's look at these same programs written for a sequential disk file. The first program writes a series of names of budget items on disk.

Functionally, this program appears the same to the user as the tape program. The program clears the screen and then prompts the user to enter an item or the word END to terminate the input. Each name is then written on the disk file. Here is the program:

```
100 REM CREATE SEQ DISK FILE
110 PRINT CHR$(147):REM CLEAR SCREEN
120 DOPEN#5,"BUDGET",DO,W
130 INPUT "ENTER ITEM NAME (END)";N$
140 IF N$="END" THEN 170
150 PRINT#5,N$
160 GOTO 130
170 PRINT#5,"END"
180 DCLOSE#5
```

There are a few subtle differences when we are creating a disk file. The first involves the use of the DOPEN to open the file, give it a name, and assign the disk drive. Once the file is opened things proceed the same as for tape until the end of the data is reached. Tape files used the status indicator (ST) to test for end of file on the input. But, since ST is not available for disk, we must create our own end of file indicator. This program simply writes an extra record, containing the word "END", on the disk file to identify the end of file. The data on the disk will look like this:

```
RENT(cr)TELEPHONE(cr)LIGHT(cr)HEAT(cr)SUPPLIES(cr)END(cr)
```

When using a special record to indicate end of file it is quite important to ensure that the choice of word or value is not something that can legitimately appear as part of the data.

Now, when this file is read the program can test for the word "END" and when it is found it will indicate end of file has been reached.

### ***Reading Budget Names from Disk***

To read the budget names file, a program similar to the tape input program is required. Again there are some minor differences. One of these changes is how to test for end of file. Here is the program.

```
200 REM READ AND DISPLAY BUDGET FILE
210 DOPEN#5,"BUDGET",DO,R
220 PRINT CHR$(147)
230 PRINT "BUDGET ITEM"
240 PRINT
250 INPUT#5,N$
260 IF N$="END" THEN 290
270 PRINT N$
280 GOTO 250
290 DCLOSE#5
```

The end of file is checked in statement 260 immediately after the input is read. If the "END" record was read then control of the program branches directly to statement 290 where the file is closed and the program terminates.

## How to Handle Multiple Fields on Disk Files

Disk files also frequently use records consisting of more than one field. This situation is handled in the same way as tape by writing a carriage return (CHR\$(13)) between each field on the record. Don't forget to separate each field with a semicolon, not a comma.

End of file on sequential disk requires a separate record as we saw in the preceding examples. But, since we are recording multiple fields each field must be given a value. In the following program the Name field (N\$) will be given the value "END" and the Amount field (A) will be given a zero value at end of file.

```
100 REM CREATE NEW FILE WITH MULTIPLE FIELDS
110 CR$=CHR$(13)
120 PRINT CHR$(147):REM CLEAR SCREEN
130 DOPEN#5,"BUDGET 2",W←
140 INPUT "ENTER ITEM NAME (END)";N$
150 IF N$="END" THEN 190
160 INPUT "ENTER AMOUNT";A
170 PRINT#5,N$;CR$;A
180 GOTO 140
190 PRINT#5,"END";CR$;0←
200 DCLOSE#5
```

defaults to drive 0

end of file record

Note the use of the variable CR\$ for the carriage return. Since carriage return codes are used more than once in the program, line 110 assigns the value once to CR\$ which then may be used as needed. This approach makes it easier to use the return code rather than defining it completely each time with the CHR\$ function.

Now let's look at a program to read this file:

```
100 REM READ AND DISPLAY BUDGET FILE
110 DOPEN#5,"BUDGET 2"←
120 PRINT CHR$(147)
130 PRINT "ITEM","AMOUNT"
140 PRINT
150 INPUT#5,N$,A
160 IF N$="END" THEN 190←
170 PRINT N$,A
180 GOTO 150
190 DCLOSE#5
```

defaults to drive 0  
and read

end of file reached

This program is really not much different from the tape program except for end of file handling, which we have already discussed. If at this point you haven't already entered these programs into your computer why not give them a try and observe first-hand how they work. Then try and rewrite them to include some other fields such as date, estimated amount, and actual amount. If you try this it will be necessary either to use a different filename or first scratch BUDGET 2 from your disk.

## Detecting Disk Errors

Occasionally an error will occur when either opening a file or reading or writing to the file. Usually the error will be the fault of the program, such as trying to write a file that already exists with the same filename. Rarely will the error be a problem with the floppy disk or the disk drive. In any case when there is an error the program needs to know about it, since to continue on blindly assuming that nothing is wrong can be disastrous. In the previous programs we did make that assumption but this is not advised for serious programs.



How do you know when an error occurs on the disk? Well, try entering the command

**? DS\$**

this command will print the disk status string, which should appear as follows:

**00, OK,00,00**

This indicates there was no error on the previous disk operation.

There is a similar variable DS that is the numeric equivalent of this message. By typing

**? DS**

the result would normally be

**0**

Any value other than zero would indicate an error of some kind had occurred during the last disk operation.

If a program attempted to open a file for output that already existed the contents of DS\$ would be

**63, FILE EXISTS,00,00**

and DS would be 63. A complete list of error codes is listed in Appendix D.

Now let's rewrite the last program, which reads and displays the budget file, and include code to check the disk status after each I/O operation.

```
100 REM READ AND DISPLAY FILE
110 DOPEN#5,"BUDGET 2"
120 IF DS<>0 THEN PRINT DS$:GOTO 210
130 PRINT CHR$(147)
140 PRINT "ITEM","AMOUNT"
150 PRINT
160 INPUT#5,N$,A
170 IF DS<>0 THEN PRINT DS$:GOTO 210
180 IF N$="END" THEN 210
190 PRINT N$,A
200 GOTO 160
210 DCLOSE#5
```

At statement 115 and 155 the program now tests DS for a zero value which would indicate all is well. However, if DS is not equal to zero then the program will print out the contents of DS\$, which will give a description of the error that occurred. The program then branches to the close statement at 190 and the program is terminated.

If the program had attempted to open file "BUDGET 1" then the error

**65, FILE NOT FOUND,00,00**

would have been displayed.

## **MAINTAINING THE CHECKBOOK**

A common need we all have is to maintain a checkbook for our deposits and withdrawals against a checking account. To write this program we need to identify how to do several things. First is the need to create a new file with an opening balance. Next is the need to be able to add additional entries because of deposits and withdrawals against the account. And third is the ability to delete entries at the end of a month or similar time period; otherwise the file will continue growing larger and larger.

Actually, these provisions are not uncommon for many types of files. For example, an accounts payable or receivable file, an inventory file, a file of students, and so on. The list is endless. Although each application will have different specific needs some of the basics developed here will not need to change very much.

The procedure for updating a sequential disk file (the checkbook) will be the same as that used for tape. In other words, this program will also read the file into memory using arrays, update the data in the arrays, and then rewrite the updated file back onto the disk. The updated file then replaces the old file, which is scratched from the disk. For this reason it is vitally important that the program be thoroughly tested since if there are any errors they may cause damage to the disk records, which cannot then be retrieved.

### ***Replacing a Current File***

How do we tell the disk that we want to replace a file instead of writing a new one? This is done by a code in the DOPEN statement. The code is an "@" used as follows:

```
100 DOPEN#3,"@CHECKFILE",D0,W
```

You will recognize this open as a disk open for file number 3 with a filename of "CHECKFILE". The file is on drive 0 and is an output file. Notice the use of the "@" immediately before the filename so the name appears as "@CHECKFILE". This use of the "@" indicates we are opening an existing file that is to be replaced with new or revised data.

### ***Using Variable File Names***

The program we are about to write also has another need. We would like to be able to accept any filename as input to the program. That way we are not limited to just one file but could access any number of possible files. This is a more realistic approach to disk since one floppy can store a number of different files. This means we need to use a variable for the filename. Here is an example of how a filename might be entered and then used in the DOPEN statement:

```
100 INPUT "ENTER FILENAME";F$
110 DOPEN#3,(F$),D0,W
```

This example uses the variable F\$ to provide the filename in the open statement. Notice that it is enclosed in parentheses. Now statement 110 will open whatever file the user specifies in response to the input query in line 100. But, if this is a file to be replaced, we are expecting the user to include the "@" as the first character of the filename. This might be acceptable but it would be much cleaner if the program could take care of this detail. Here's how it could be done:

```
100 INPUT "ENTER FILENAME";F$
110 DOPEN#3,("@"+F$),D0,W
```

Now line 110 concatenates the "@" to the filename and the user need only enter the name of the file, which is a more natural response.

## **English Code**

Having completed these preliminary discussions we can now begin developing the program logic. For this program we will once again use English code to develop our logic.

This program makes quite extensive use of subroutines—an approach you should be taking by now in all of your programs. As much as possible strive for generality in all of your subroutines so they may be used in other programs as the need arises. Later it will become evident that the solution we have developed here could be made even more general.

### **Start-up Menu**

1. Create New Checkbook File
2. Update Checkbook File
3. Stop

### **Create New Checkbook File Subroutine**

1. Accept Filename
2. Accept Date and Opening Balance
3. Write Entry on New File

### **Update Checkbook File Subroutine**

1. Accept Filename
2. Load File into Arrays
3. Add, Change Menu Until Done
  - 3.1 If Add Call Add Data to Checkbook
  - 3.2 If Change Call Change Data on Checkbook
4. Rewrite Updated File on Disk
5. Return

### **Add Data to Checkbook Subroutine**

1. Accept Data Until Done
  - 1.1 Accept Date, Desc, With/Dep
  - 1.2 Append to End of Table
  - 1.3 Calculate and Display New Balance
  - 1.4 If Table Full Display \*Warning\*
2. Return

### **Change Data on Checkbook Subroutine**

1. Display Items Until End of Table
  - 1.1 If Screen Full (20 items) then
    - 1.1.1 Proceed or Change Items?  
If Change Call Accept Change or Delete
  - 1.2 Proceed to Next Screen

### **Accept Changes and Deletions Subroutine**

1. If Change Then Call Change Entry
2. If Delete Then Call Delete Entries
3. Return

## Change Entry

1. Accept Entry Number
2. Change Item
3. Readjust Balance
4. Return

## Delete Entries

1. Accept Entry Number
2. Move Following Items Back in Array
3. Reset End of Table
4. Display Screen Page with Changes
5. Return

The English code at this level is still quite general. For instance, it doesn't go into detail about how each menu is going to appear in the program. Neither does it specify exactly what subscripting will be used when referencing array elements. This next level of detail will be developed in the program code itself.

However, there are a few places in the solution that may not be at all clear at this stage. For example, in the 'Change Data on Checkbook Subroutine', reference is made to displaying a screen of data. Also the 'Delete Entries' subroutine makes a casual reference to moving items back in the array. Both of these statements need clarification.

### ***Display Screen Page***

Checkbooks normally have pages with deposits and withdrawals shown and the related balance, date, and description of the entry. In this program we would like to treat the screen display as a page of data rather than one line at a time. To do this we need to display contents of the arrays until one page is shown. Then changes to that page can be accepted.

Figure 10.3 shows how lines from the array are displayed to create one page of output. At the end of the page is a prompt asking if there are any changes to be made to this page or if the user wishes to proceed to the next page. If a 'C' is entered then a new prompt asks for the type of change, while a 'P' response would cause the next page to be displayed on the screen.

The variable LI is used to count the number of lines displayed on the screen. When LI has reached 20 then the loop is put on hold temporarily until the user can respond to the prompt. When the command to proceed is given the program then resets LI to zero, restarts the loop, and displays the next 20 lines. At some point there may be less than 20 lines left to display. In this case the loop will terminate with fewer than 20 lines on the page.

### ***Deleting Table Entries***

Another interesting problem to solve is the deleting of items from the table. This might be done at the end of a month when last month's items are deleted—or if not last month's items then items from several months or even a year ago. It is the user's choice.

Assuming entries will be made in date sequence the program will be designed to delete all entries from a specified position on the page back to the beginning of the checkbook. The user specifies the position by entering the line number of the entry that has been displayed with the other checkbook fields. Figure 10.4 shows the procedure for deleting table items.

If the user response indicates that items 1 to 4 are to be deleted then all items from item 5 to the end of the data (NE) are shifted back 4 positions in the arrays. In other words, item 5 is moved back to position 1, item 6 to position 2, item 7 to 3, and so on.

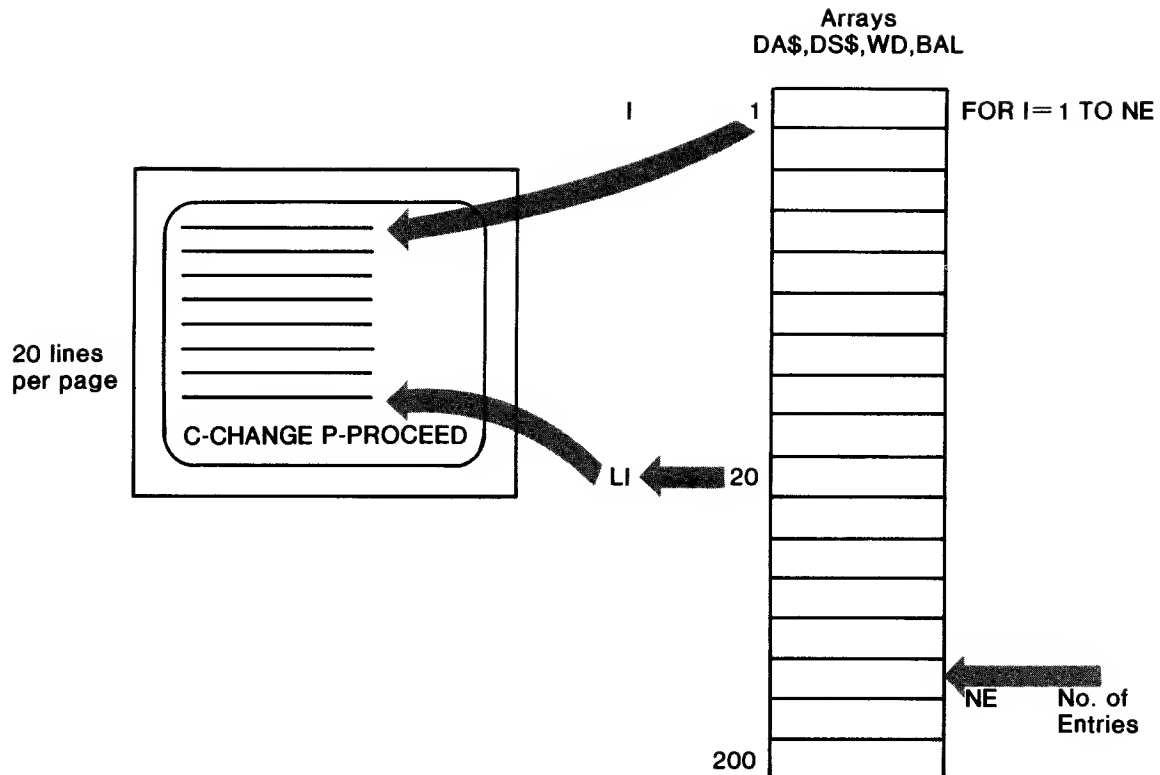


Figure 10.3 Display screen page

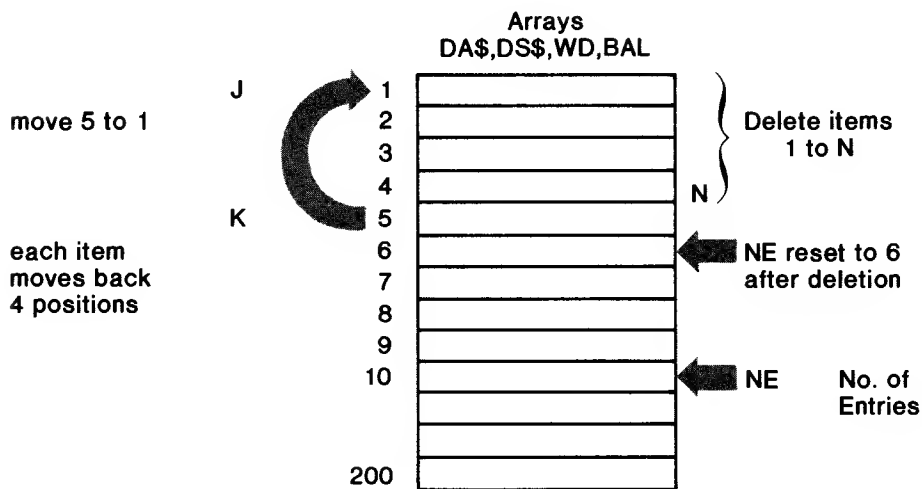


Figure 10.4 Deleting four table items

When all data up to position NE have been moved, the deletion is complete. Then NE is reduced by 4 (since four items were deleted) and is now at position 6, referring to the current number of entries in the table. One problem with this technique is that the more data the arrays contain, the more time the program will take to do a deletion. Using linked lists would improve this time factor considerably.

The complete checkbook program is shown in figure 10.5.

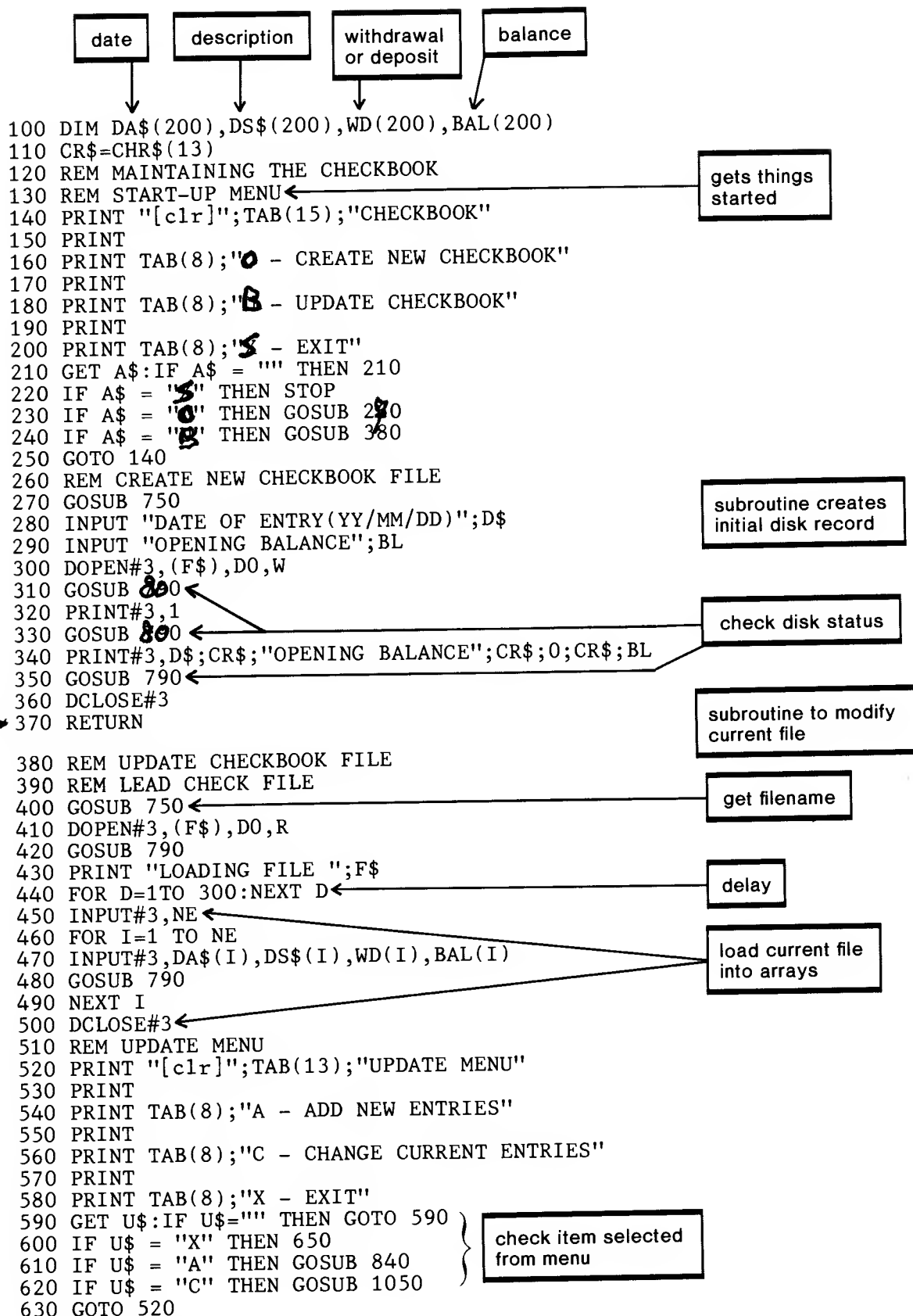


Figure 10.5 Checkbook update program

```

640 REM SAVE UPDATED FILE ON DISK
650 PRINT "REWRITING FILE ";F$←
660 DOPEN#3,("@"+F$),D0,W
670 PRINT#3,NE
680 GOSUB 790
690 FOR I=1 TO NE
700 PRINT#3,DA$(I);CR$;DS$(I);CR$;WD(I);CR$;BAL(I)
710 GOSUB 790
720 NEXT I
730 DCLOSE#3←
740 RETURN
750 REM ACCEPT FILENAME←
760 PRINT "[clr]";
770 INPUT "ENTER FILENAME";F$
780 RETURN
790 REM CHECK DISK STATUS←
800 IF DS=0 THEN RETURN
810 PRINT DS$
820 DCLOSE#3
830 STOP

840 REM ADD DATA TO CHECKBOOK
850 PRINT "[clr]"; TAB(13);"ENTER NEW DATA"
860 PRINT
870 PRINT "CURRENT BALANCE IS ";BAL(NE)
880 PRINT
890 PRINT "ENTER DATA AS FOLLOWS:"
900 PRINT
910 INPUT "DATE(YY/MM/DD) OR (QUIT)";D$
920 IF D$="QUIT" THEN 1040
930 NE=NE+1
940 DA$(NE)=D$
950 INPUT "DESCRIPTION";DS$(NE)
960 INPUT "DEPOSIT 0[3 1t's]";D←
970 INPUT "WITHDRAWAL 0[3 1t's]";W←
980 WD(NE)=D-W
990 BAL(NE)=BAL(NE-1)+WD(NE)
1000 PRINT
1010 PRINT "NEW BALANCE = ";BAL(NE)
1020 IF NE = 200 THEN PRINT "*WARNING* - TABLE FULL"
1030 GOTO 900
1040 RETURN

1050 REM CHANGE DATA ON CHECKBOOK
1060 PRINT "[clr]ENTRY DATE","DESC","WITH/DEP","BALANCE"
1070 PRINT
1080 FOR I=1 TO NE←
1090 PRINT I;" ";DA$(I);TAB(15);LEFT$(DS$(I),10);TAB(28);WD(I),BAL(I)
1100 LI=LI+1
1110 IF LI<20 THEN 1130←
1120 GOSUB 1150
1130 NEXT I
1140 GOSUB 1150
1150 REM ACCEPT CHANGES TO THIS PAGE
1160 IF LI=0 THEN RETURN
1170 PRINT
1180 PRINT "C - CHANGE OR P - PROCEED"
1190 GET P$:IF P$="" THEN 1190
1200 IF P$="C" THEN GOSUB 1230
1210 LI=0←
1220 RETURN

```

after updating is finished  
rewrite file to disk

subroutine to get filename

subroutine to check  
disk I/O status

subroutine to place new data  
in checkbook file

permits "0" entry  
by just pressing Return

determine net,  
withdrawal, or deposit

subroutine to change  
existing entries

display  
current data

stop display after 20 lines  
for changes

subroutine to query changes  
for current page on display

if no change (P-response) or changes  
finished, then clear line counter and  
return to display next page

Figure 10.5 Checkbook update program (continued)

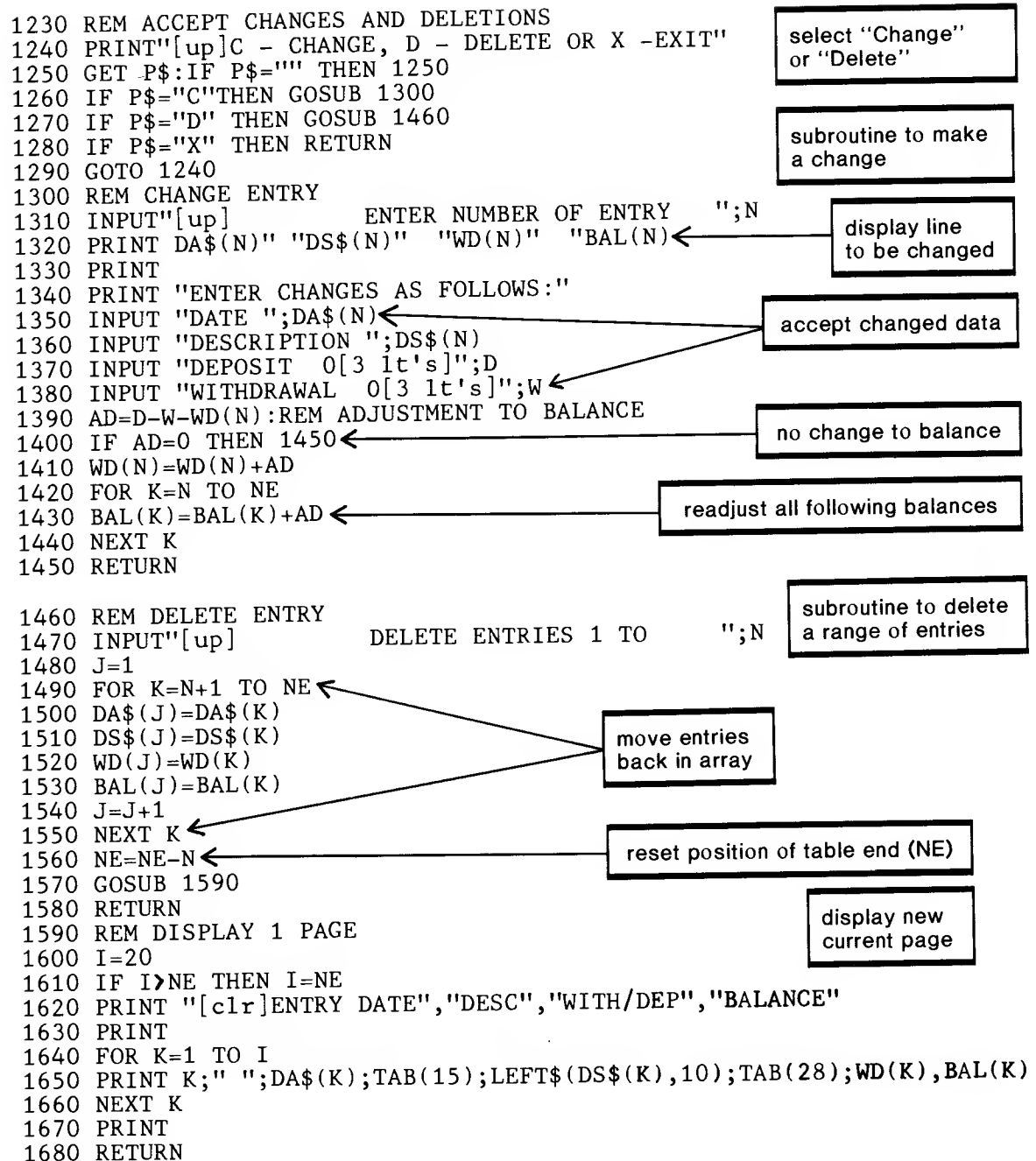


Figure 10.5 Checkbook update program (continued)

## RELATIVE ACCESS FILES

While sequential disk files have a speed advantage over tape, relative files have an even greater benefit. As we already know, a sequential disk file must be read sequentially from beginning to end. If our files are quite short there is no apparent disadvantage to this. However, if a file consists of hundreds or even thousands of records, then waiting for the 2999th record can be time-consuming even on a sequential disk file. This is when relative files come to the rescue!

The primary advantage of a relative file (also called a random access file) is the ability to access any record in the file directly, without reading any other records. So the program can read the 2999th record as the first record in just a fraction of a second.



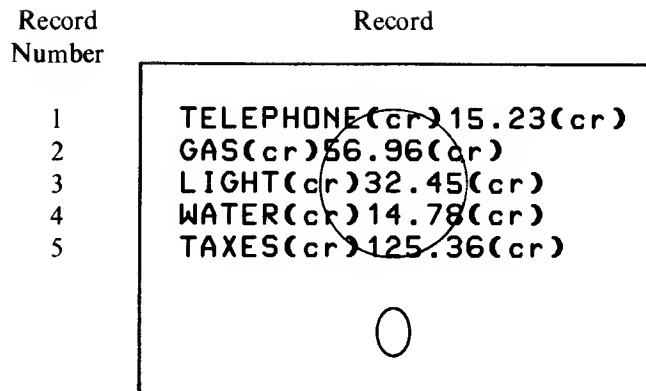


Figure 10.6 Relative file organization

Figure 10.6 shows a symbolic representation of a relative file. Records in a relative file are numbered consecutively from 1 to the last record in the file. To access a given record it is only necessary to specify the record number desired, and the disk will supply that record to your program.

Relative files are opened much like any other disk file, the one exception being the use of a length parameter. This parameter determines the length of record to be used in the file and is necessary for DOS to find any specific record you may ask for. Here is the general format for the DOPEN when used with relative files.

DOPEN#fn,"filename",Lx,Dn
<p>where fn refers to the file number.</p> <p>"filename" is the name used to identify the relative file.</p> <p>Lx L refers to the maximum record length (x) in the file, including any carriage return characters. Since records can contain data such as names, addresses, and descriptions that are variable in length the maximum length that all records will need must be determined. The size permitted may be from 1 to 254 bytes.</p> <p>Dn D refers to Drive and n to the drive number. If the disk is on drive 1 then the entry is D1. If drive 0 then use D0 or simply leave this entry out and it will default to drive 0.</p>

Here is an example of a DOPEN to use when creating a new file.

```
200 DOPEN#5,"INVENTORY",L20,D0
```

↑ ↑ ↑ ↑  
filename filename record drive  
  length

In this case a file named INVENTORY is opened as file number 5. The floppy disk will be in drive 0 and the relative file will consist of 20 byte records. You may write records that are less than 20 bytes but may not exceed this length.

The following DOPEN would be used for accessing the above file after it has been created.

```
200 DOPEN#5,"INVENTORY"
```

↑ ↑  
filename filename

For this open, only the filename is required since the length has already been established on the file. The drive number will default to 0 and so reference to attribute D0 is unnecessary.

## Creating a Relative File

A relative file is created in much the same way as a sequential file. The main difference is the length attribute in the DOPEN statement. Once the file has been opened data is printed to it in the same way as when we created a sequential disk file.

The following program creates a relative file called BUDGET.REL on file number 5. The use of .REL at the end of the filename is not mandatory for a relative file but is a way of reminding yourself that this is a relative file.

```
100 REM CREATE RELATIVE FILE
110 CR$=CHR$(13)←carriage return
120 PRINT CHR$(147):REM CLEAR SCREEN
130 DOPEN#5,"BUDGET.REL",L30←record length 30
140 INPUT "ENTER ITEM NAME (END)";N$
150 IF N$="END" THEN 190
160 INPUT "ENTER AMOUNT";A
170 PRINT#5,N$;CR$;A
180 GOTO 140
190 DCLOSE#5
```

This program will create the file shown in figure 10.6, provided that the data is entered in response to each input prompt. The relative file is opened with 30-byte records. At this time BASIC is prepared to reference record 1. When the first PRINT# is done in statement 170 the data written to the disk will be automatically referenced as record number 1. When the loop gets back to statement 170 a second time record 2 is written and so on. Notice that you do not need to count the records in the program.

Two potential problems can occur in this program. First, it is up to the user to ensure records do not exceed 30 bytes; otherwise some very nasty things will happen to the file. A complete program should actually check the length of the data before writing it to the disk, but for simplicity's sake we are not doing that checking here.

A second possible problem may occur since we are not doing the disk status (DS) check after each disk operation. Normally this should also be done in a disk program.

## Accessing a Relative File

Reading records from a relative file is almost as easy as reading a sequential file. The only difference is that a relative file needs to know what record you want. This is done with the RECORD# statement. RECORD# has the following format:

RECORD# f n, n  
          ↑          ↑  
      filenumber  record  
                  number

This statement normally precedes the INPUT# statement that reads the record from the relative file. For example,

```
200 RECORD#4,12
210 INPUT#4,N$,A$
```

would set file number 4 to the 12th record in statement 200 and then read the 12th record in statement 210. It's as simple as that.

To read the 2999th record you could use:

```
200 RECORD#4,2999
```

Now here is a program to read record 3 of the BUDGET.REL file we created earlier.

```
100 REM READ RELATIVE FILE
110 DOPEN#5,"BUDGET.REL"
120 PRINT CHR$(147)
130 PRINT "ITEM","AMOUNT"
140 PRINT
150 RECORD#5,3
160 INPUT#5,N$,A
170 PRINT N$,A
190 DCLOSE#5
```

set to record 3

input record 3

The relative file is opened in statement 110. There is no reference this time to record length since this has already been established in the file. The disk drive defaults to 0 and the file will be considered read/write, meaning we can either read from it or write on the file.

Statement 150 supplies the record number to be read and the following statement (160) inputs the record. The RECORD# statement can also use a variable, since we don't always want to read record 3 or whatever. Here is an example of a variable in the RECORD# statement:

```
400 RECORD#4,(I)
```

When the variable is used it must be enclosed in parentheses. This suggests that you could use an expression to compute the record number.

The previous program could be rewritten to accept a record number from the user (statement 145) and then use this number to access the desired record in statement 150.

```
100 REM READ RELATIVE FILE
110 DOPEN#5,"BUDGET.REL"
120 PRINT CHR$(147)
130 PRINT "ITEM","AMOUNT"
140 PRINT
145 INPUT "ENTER RECORD NUMBER";R
150 RECORD#5,(R)
160 INPUT#5,N$,A
170 PRINT N$,A
180 DCLOSE#5
```

### **Updating a Relative File**

If you went through the program for updating a sequential file the process for updating a relative file should prove to be a pleasant surprise. First, remember that with relative files we are working with just the record that needs updating. Therefore we can ignore all of the other records on the file. No searching for the right record and no rewriting the complete file is necessary.

The steps to be taken for updating a relative file are as follows:

1. Get the Record Number
2. Issue a RECORD# command
3. Input the record
4. Update the record
5. Issue a RECORD# command
6. Rewrite the record

Steps 1 to 3 should be self-explanatory at this point since they are the same steps for accessing a record. Step 4, updating the record, is program code to accept changes to the existing data in the record by modifying one or more of the variables that contain the data.

Step 5 may not seem necessary since we already have issued a RECORD# command in step 2. However, it is important since DOS will increment the record number it is using and move on to the next record, which, of course, we don't want to update. The last step is simply to rewrite the record using a PRINT# statement as if we were creating a new record. This new record will then replace the old one on the relative file.

Here is the program:

```

100 REM UPDATE RELATIVE FILE
110 DOPEN#5,"BUDGET.REL"
120 PRINT CHR$(147)
130 INPUT "ENTER RECORD NUMBER(0 - TO QUIT)";R
140 IF R=0 THEN 230
150 RECORD#5,(R) } ← read the record
160 INPUT#5,N$,A }
170 PRINT "TYPE CHANGE OR PRESS RETURN"
180 PRINT "ITEM - ";N$:INPUT "[up 9 rt's]";N$ } ← accept
190 PRINT "AMOUNT - ";A:INPUT "[up 9 rt's]";A } changes
200 RECORD#5,(R)
210 PRINT#5,N$;CHR$(13);A } ← rewrite the updated record
220 GOTO 130
230 DCLOSE#5

```

An interesting technique is used in statements 180 and 190 to ease the burden of retyping data that does not need to be changed. First the message "ITEM—" is displayed with the contents of N\$ which is the name of the item. Then the INPUT statement moves the cursor up to the line just printed, and then to the right 9 columns so that the cursor is over the first character of the data. If there is no change to N\$ then by simply pressing return the data is reentered without any change. A change can be made by simply typing over the old value on the screen and pressing return. Now the new value will be in N\$.

### Appending Records to an Existing File

So far we have written programs that assume that all records on the relative file are created in the beginning of the file's life. This is hardly realistic. Many applications require the ability to append (add) records to the file at any time. An inventory file will most certainly be stocking new items at some time in the future, a book list will have new books added to it, a student record system will need to be able to add new records to the file.

Appending a new record to a relative file is no different from creating a new file, except in this case you don't specify the record length in the open statement. The open is handled in the same way as for file updating. What you need to know is the record number of the new record to be appended to the file. And that is the catch!

How do you know the number of records in the file? So far we don't. What is necessary is a means for determining the number of records (N) currently in the file. Once you know this then the new record is record number  $N + 1$ .

One way to identify the number of records in the file is to retain a record as part of the file where you can count each record as it is stored on the file initially. Figure 10.7 shows the budget file created in this way.

All of the records in the file are the same as before except they are one record-number higher. In record 1 is a single numeric value, which is currently 6, to indicate there are six records in the file. When a new record is added to the file record 1 can tell us that it should be record 7 from  $6 + 1$ .

Record Number	Record
1	6
2	TELEPHONE(cr) 15.23(cr)
3	GAS(cr) 36.96(cr)
4	LIGHT(cr) 32.45(cr)
5	WATER(cr) 14.78(cr)
6	TAXES(cr) 125.36(cr)
	0

**Figure 10.7** Relative file with a record count in record 1

A program to use this information from record 1 and append new records to the file is shown in figure 10.8. This is a revision of the update program with the capability now of adding new records to the file. Notice that each reference to the file uses subroutine 510 to check the disk status in the event of an I/O error.

As soon as the file is opened the record number is read into variable N from record 1 in statements 130 and 140. The first time this file is used N will contain the value 6 as we saw in the previous diagram.

The update part of the program is the same as before so look down at the append subroutine in statements 400 to 500.

To add a new record, N is increased by 1 in statement 430 and then in 440 and 450 the new record is written as record N (now 7). Now since we have seven records on the file, record number one must also be updated. So in statements 470 and 480 record 1 gets written with the new value of N and we now have seven records counted on the file. Any number of records may be added using this method, up to the capacity of the disk.

### ***How to Use an Index File***

One problem with the previous series of programs is the assumption that we will always know the record number of the record we want. This may be valid for some applications, for instance when a small number of items are used, or if the identification numbers are identical to the record number.

However, in many cases there are a large number of items in a file and their identification may not correspond to the record number. Take, for example, a retail store that sells indoor plumbing fixtures. There may be several hundred different items in inventory, let's say 248 to be exact. These fixtures are not likely identified by the numbers 1 to 248 but by some other part number. If this part number is a four-digit number it would suggest we need to plan a relative file with 9999 possible records, many more than are really necessary for the file.

This problem is resolved by having only 248 records for the relative inventory file but having a second, sequential file called an index file. The index file (figure 10.9) is simply a list of the part numbers, which are read into an array. Element 0, which gets the first record from the index file, tells us how many records are in the relative file. The remaining elements of the array contain part numbers. For instance, part number 2377 in element 1 indicates that this part is stored in record 1 of the relative file. Part number 2105 points to record 4 since it is in element 4 of the index.

To find a part in the file the user of the program would first respond to a prompt which asks for the part number.

ENTER PART NUMBER ?  
2213 ← PN

```

100 REM **          RELATIVE FILE **
110 DOPEN#5,"NUM.REL"
120 GOSUB 510
130 RECORD#5,1
140 INPUT#5,N←
150 GOSUB 510←
160 PRINT CHR$(147)
170 PRINT TAB(15);"N - NEW RECORD"
180 PRINT
190 PRINT TAB(15);"U - UPDATE"
200 PRINT
210 PRINT TAB(15);"X - EXIT"
220 PRINT:PRINT
230 GET A$:IF A$="" THEN 230
240 IF A$="N" THEN GOSUB 400←
250 IF A$="U" THEN GOSUB 290←
260 IF A$="X" THEN 540
270 GOTO 160
280 REM ** UPDATE RECORD **
290 INPUT "ENTER RECORD NUMBER ";R
300 RECORD#5,(R)
310 INPUT#5,N$,A
320 GOSUB 510
330 PRINT "TYPE CHANGE OR PRESS RETURN"
340 PRINT "ITEM    - ";N$:INPUT "[up 9 rt's]";N$
350 PRINT "AMOUNT - ";A:INPUT "[up 9 rt's]";A
360 RECORD#5,(R)
370 PRINT#5,N$;CHR$(13);A
380 GOSUB 510
390 RETURN
400 REM ** APPEND NEW RECORD **
410 INPUT "ITEM    - ";N$
420 INPUT "AMOUNT - ";A
430 N=N+1
440 RECORD#5,(N)
450 PRINT#5,N$;CHR$(13);A←
460 GOSUB 510
470 RECORD#5,1
480 PRINT#5,N←
490 GOSUB 510
500 RETURN
510 REM ** CHECK DISK STATUS **
520 IF DS=0 THEN RETURN
530 PRINT DS$
540 DCLOSE#5
550 STOP

```

get number  
of records

check disk status

append new subroutine

update subroutine

from record  
1

append new record

revise number  
of records

Figure 10.8 Adding records to a relative file

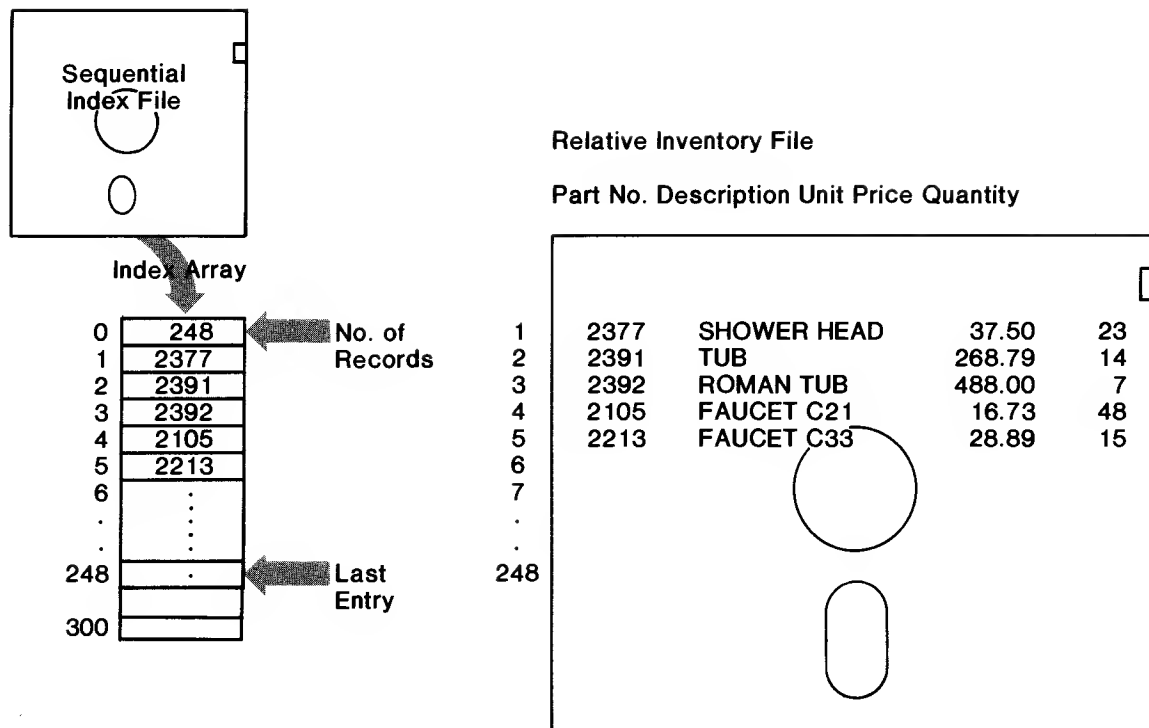


Figure 10.9 Using an index file

The program then looks up the part number in the index table and gets the appropriate record. This is the type of code that could be used:

```

100 REM SEARCH INDEX TABLE (IN)
110 FOR I=1 TO N
120 IF PN=IN(I) THEN 160
130 NEXT I
140 PRINT "PART NUMBER ";PN;" NOT FOUND"
150 RETURN
160 RECORD#5,(I)
170 INPUT#5,PN,DS$,UP,QT
180 RETURN

```

**N is 248**

**get relative record**

It is possible, using the method of an index file, to use a description to identify records in the relative file. In this case an alphanumeric array containing descriptions would be used. A look-up of the description would then point to the appropriate relative record.

## REVIEW QUESTIONS—CHAPTER 10

1. Describe the organization of a floppy disk.
2. What kinds of files may be used on disk? Discuss the pros and cons of each.
3. Describe open and close statements for the disk. What defaults may be used in the open?
4. How is end of file handled on a sequential disk file?
5. What is the purpose of DS and DS\$? How are they used for disk applications?

6. The questions that follow refer to the checkbook program in this chapter.
  - a) Examine the routine that deletes entries from the checkbook. What happens to the balance when a delete occurs?
  - b) Subroutine 1300 accepts changes to entries but requires that all of the data be retyped even if only one component is changed. How could this subroutine be changed so that only changes need to be retyped?
  - c) When more than 20 items are in the checkbook the program scrolls the screen to the next 20 items. How could these begin with a new screen and display from the top?
7. What is the purpose of the RECORD statement when using relative files?
8. Why is an index file often necessary for use with a relative file?

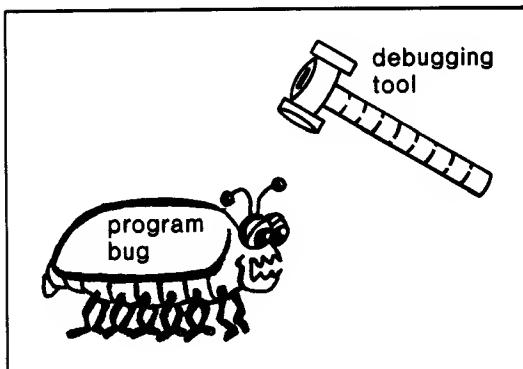




# 11

## ***How to Debug Your Programs***

**T**hroughout this book we have spent a lot of time writing BASIC programs for the PET/CBM using many of the features of the BASIC 4.0 interpreter. In every case the programs we have used were correctly written (I hope!) and would run as expected. But is this realistic? Admittedly, some of the programs did not work the first time and required changes and corrections until they did what was intended. The process of finding and correcting errors is known as debugging the program.

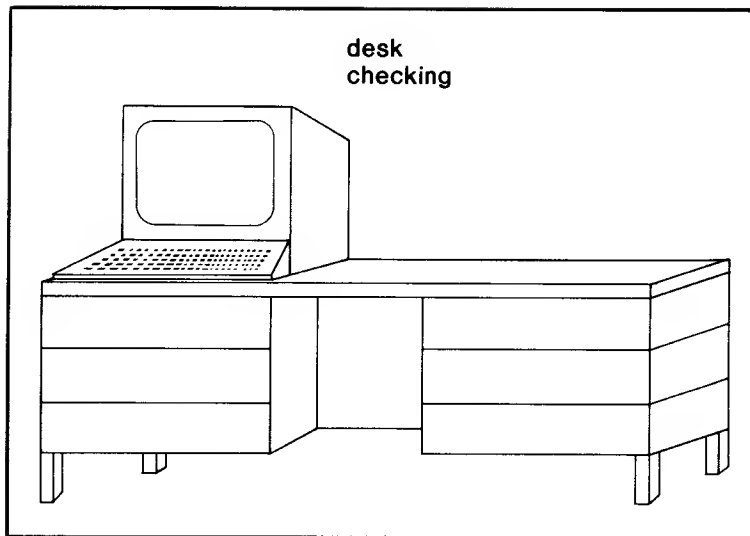


To debug a program successfully requires the development of new skills to supplement your programming skills. By following certain predefined steps, a program may be debugged and made to work in the way that you originally intended.

Here are the main things we will consider when debugging a program:

1. Desk Checking
2. Syntax Errors
3. Test Data Preparation
4. Immediate Mode Debugging
5. Tracing Program Logic

Generally, testing proceeds in this order, but changes to the program at any time may require you to go back to any of the earlier steps. This may be the result of errors that were overlooked previously or because of new errors that were introduced during the debugging process.



## **DESK CHECKING**

This is probably the easiest of all debugging techniques and yet the most often overlooked. First use one of the program planning techniques we discussed. Either a flowchart or English code will help in the planning stage. It has been proven that programmers who take the time to plan will have substantially fewer errors in their programs.

After careful planning of the solution, write the PET BASIC code. Unless the program is very simple it should be done on paper, not at the PET or CBM. At this point, take some time to desk check the code by simply reading what you have written. It is surprising how many errors can be detected at this stage without the computer's assistance. Types of errors to look for at this stage are spelling errors, logic errors, missing or duplicate statements, and statements out of order.

## **SYNTAX ERRORS**

In a formal sense syntax refers to improper punctuation of statements in the language whereas another term, semantics, refers to the proper understanding of the statement format. This is generally discussed throughout the book, so a detailed presentation will not be given here.

Basically if you take care to follow the statement formats your program will be correct as far as BASIC 4.0 is concerned. If you make a mistake the PET/CBM will give you an error message when you attempt to RUN the program. For example, the statement:

```
10 FOR I=1,10
```

may sound OK, but when the program is run the error message

```
?SYNTAX ERROR IN 10
```

will be displayed on the PET's screen. The statement's syntax is incorrect because the format of a FOR statement requires the keyword TO instead of a comma between the 1 and 10. Therefore the statement:

```
10 FOR I=1 TO 10
```

is the correct one.

Syntax errors are many and varied. In a number of cases, it is difficult to determine what has caused the error. Although there are precise definitions of errors, we are more concerned with finding and correcting the errors. The following is a list of errors that are encountered quite frequently by programmers. Most of these errors are identified by error messages from the BASIC interpreter.

Type of Error	Example	Explanation
Punctuation	10 N(I K)=A+B	The comma separating subscripts I and K is missing. Two-dimensional arrays require two subscripts separated by a comma.
	10 PRINT A B	Two or more variables in input and output statements must be separated by commas. PET BASIC treats this as variable AB rather than an error so this type of error is easy to miss.
Parentheses	10 N=N+1)*5 20 A=(K+1*5+(J/N)))	Either too few brackets (statement 10) or too many (20) can be the cause of this problem. BASIC 4.0 flags these as syntax errors. Make sure that all brackets are matching pairs.
Operator	10 N+N=K	The equal operator is on the wrong side of the expression.
	50 J=B(N+L)	The multiply operator is missing between B and the parenthesis. PET assumes B to be an array with (N+L) the subscript; so again this is an error that is easily overlooked.
	70 K=P* /J	In this example two operators are used together; a variable is probably missing.
Mismatch	200 N="DON CASSEL"	This occurs when an alphanumeric value is assigned to a numeric variable or vice versa. PET calls this a Type Mismatch Error.
Illegal Verb	10 S=SQRT(256)	An illegal verb is also a class of syntax error. This happens when a keyword is misspelled. SQRT is not legal in PET BASIC.

During program execution, other errors, which are not indicated earlier, may be found. These are generally self-explanatory errors, such as dividing by zero, trying to calculate the square root of a negative number, and having arithmetic overflows or subscripts out of range.

For example, in the following case in which a constant is being used:

```
10 DIM NO(15)
:
:
80 NO(21)=T * K
```

it is evident that the subscript of 21 is the culprit; maybe it should have been 12, or possibly NO should have been dimensioned with more elements. However, if the statement in error had been:

```
80 NO(I)=T * K
```

then it would be necessary to trace the values of I to see why it had exceeded the permissible range of the subscript.

## TEST DATA PREPARATION

The first rule of data preparation is that quality and not quantity is what counts. Beginning programmers often prepare a lot of data that looks impressive but is actually quite meaningless. Each item of data should have a specific purpose and therefore test for a specific potential problem. Some of the things for which data should be prepared are:

1. Sequence error—for tape or disk files.
2. Missing data.

3. Positive and negative values.
4. Valid and invalid codes.
5. Numbers within a specific range including the first and last numbers of the range.
6. Too much or too little data when using tables.
7. Reasonableness. An hourly pay rate over \$100.00 is probably unreasonable.
8. Length. Fields such as phone numbers, account numbers, and social insurance numbers are of predefined lengths and can be checked.

Not every program will need to test for all of these items, and some programs will need to test for other factors. The key is to be aware of possible sources of error and to consider them when testing your programs.

A final consideration is that test data should be supplied to check every statement of code written in the program. Don't assume that the program code will work just because it is obvious or simple. Errors do not happen solely in complex code; they often occur in the most unlikely places.

### **IMMEDIATE MODE DEBUGGING**

One of the advantages of using a microcomputer like the PET/CBM is the capability of examining the contents of program variables directly on the screen. Earlier in the book we discussed the use of the computer as a sophisticated calculator using immediate mode. This same mode can be used to examine the contents of the program's variables while debugging the program.

Immediate mode debugging is particularly useful if the program terminates prematurely. For instance, you are running a test of the following program:

```
10 DIM A$(100)
20 B$="STRING "
30 FOR I=1 TO 100
40 B$=B$+B$
50 A$(I)=B$
60 NEXT I
```

and the program stops with the message

```
?STRING TOO LONG ERROR IN 40
```

Now from the program code we decide that B\$, which is in statement 40, should contain the string "STRING STRING". At least that's what we intended. This can be immediately confirmed or denied by using the immediate mode to display the contents of B\$ as follows:

```
? B$
```

The PET responds with the following display:

```
STRING STRING STRING STRING STRING STRIN
G STRING STRING STRING STRING STRING STR
ING STRING STRING STRING STRING STRING S
TRING STRING STRING STRING STRING STRING
  STRING STRING STRING STRING STRING STRI
NG STRING STRING STRING
```

This is obviously not what we wanted. We can also examine I's value:

```
? I
6
```

This tells us that the loop had executed six times when the program terminated. We could try a lot of other things at this point. For instance, we could change the FOR loop to repeat only three times and then check B\$'s value. It would contain fewer repetitions but still more than the two required.

By now we should have a good idea of what is wrong: the variable B\$ is being affected by the FOR loop when our expectation was that it would be assigned the string "STRING" concatenated to itself only once. The problem is that statement 40 should precede the loop and not be contained within it.

The program is changed and now appears as follows:

```
10 DIM A$(100)
20 B$="STRING "
30 B$=B$+B$←
40 FOR I=1 TO 100←
50 A$(I)=B$
60 NEXT I
```

these two statements are now reversed

## TRACING PROGRAM LOGIC

Tracing is the process of following the program logic step by step to locate an error. This can be done manually for relatively easy problems, or program statements can be used to show the flow of the logic. The key is to know what to expect from your program. If you know what output the program is intended to produce, then when it does not produce this output, a trace can be used to determine why.

### Manual Tracing

Manual tracing is done with a copy of the program listing (so you need a printer), its output (if any), and a pencil. This method is particularly useful for small programs, simple bugs, or when there is no output to check.

**Fahrenheit-Celsius Program** To show how this works, let's write a program to display a chart of Celsius and Fahrenheit temperatures. Actually, the chart may be one we eventually want to print but it is a good debugging technique to display it first. This can save a lot of paper.

The chart is to include Celsius temperatures from 10 to 34 and their equivalent Fahrenheit temperatures as integer values. We expect the output to appear as follows:

C	F	C	F	C	F	C	F	C	F
10	50	11	51	12	53	13	55	14	57
15	59	16	60	17	62	18	64	19	66
20	68	21	69	22	71	23	73	24	75
25	77	26	78	27	80	28	82	29	84
30	86	31	87	32	89	33	91	34	93

The program written to solve this problem is as follows:

```

5  PRINT " ";
10 FOR I=1 TO 5
20 PRINT "C  F  ";
30 NEXT I
40 PRINT
50 FOR I=10 TO 30 STEP 5
60 FOR C=1 TO 4
70 F=INT((C*9)/5+32)
80 PRINT C;F;
90 NEXT C
100 PRINT
110 NEXT I

```

Annotations:

- Line 20: create heading
- Line 40: start of each line
- Lines 70-80: C = Celsius, F = Fahrenheit

When this program is run, the output produced is:

C	F	C	F	C	F	C	F	C	F
10	50								
15	59								
20	68								
25	77								
30	86								

Two problems are evident from this output; first we are getting only one column of Celsius and Fahrenheit temperatures; second there are no spaces between the lines of output as indicated in the expected results.

The second problem may be easier to solve at this point. The print statement

```
100 PRINT
```

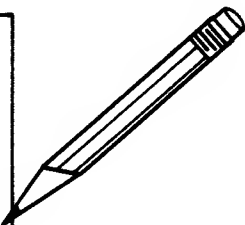
starts each new line of output but does not produce a blank line. This can be achieved by using another print statement on line 100 as follows:

```
100 PRINT:PRINT
```

To locate the problem of missing output, we will perform a manual trace of the program. To do this, we begin by writing down the names of the variables that are used in this part of the logic. Since the heading looks OK, we do not need to trace this part of the program.

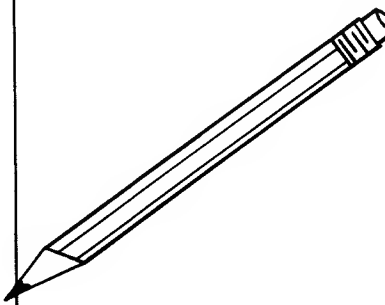
The variables that might cause the problem are I, C, and the calculation for Fahrenheit in the statement 70. We will trace the FOR loop to the NEXT at 110. The variables are written down as follows with an additional column that counts the number of iterations through the logic. The first time through the program code, the manual trace looks like this:

Iterations	I	C	F
1	10	10	50



Each time that we follow the code visually, we write down on the trace the effect that the code had on each variable. But be careful to do what the program says, not what we think it says. As we continue, the trace will look like this:

Iterations	I	C	F
1	10	10	50
2	15	15	59
3	20	20	68
4	25	25	77
5	30	30	86



At this point the program ends. It is not absolutely essential to continue the trace to the bitter end but only long enough to determine what is causing the error in the logic. By reading this trace, it is evident that C is not taking on all of the expected values. Since C represents Celsius it should take on each Celsius value. Instead it only takes on the same values as I. If we check the statement that gives C its value, we might find the error. This is statement:

```
60 FOR C=I TO 4
```

From this expression we can see that C starts at I and goes to 4. Well, if I is 10 then the loop is finished the first time, since 10 already exceeds the value 4. The FOR should start at I's value and go on for 4 more values to finish the line. In other words when I is 10, C should create temperatures 10, 11, 12, 13 and 14. Thus the statement should be:

```
60 FOR C=I TO I+4
```

The program now looks like this with the changes:

```

5  PRINT " ";
10 FOR I=1 TO 5
20 PRINT "C    F    ";
30 NEXT I
40 PRINT
50 FOR I=10 TO 30 STEP 5
60 FOR C=I TO I+4 ← revised
70 F=INT((C*9)/5+32)
80 PRINT C;F;
90 NEXT C
100 PRINT:PRINT ← extra print
110 NEXT I

```



And the correct output is:

C	F	C	F	C	F	C	F	C	F
10	50	11	51	12	53	13	55	14	57
15	59	16	60	17	62	18	64	19	66
20	68	21	69	22	71	23	73	24	75
25	77	26	78	27	80	28	82	29	84
30	86	31	87	32	89	33	91	34	93

### ***Automatic Program Tracing***

Many programs are either too large or too complex for manual tracing except in the most limited circumstances. In such a case, it is helpful if the program can be used to perform its own tracing. This can be accomplished by placing PRINT statements at temporary locations within the program.

One use of this method is to determine whether the program reaches a specific location. This can be done by placing a print statement such as

```
PRINT "TAX ROUTINE"
```

at the beginning of the program code to be tested. If the literal TAX ROUTINE displays on the screen, we know that the program does indeed reach this point. It is important to use a message that will not be confused with normal program output. Sometimes reverse characters are useful for tracing. For example:

```
PRINT"[r\5]TAX ROUTINE[r\5 off]"
```

When we are certain that the program is executing correctly or that the error has been found, the print statement will be removed since it has served its purpose.

Another use for this type of statement is to determine the contents of variables as the program is executing. This is similar to manual tracing except that it is displayed on the screen by the program. A trace statement might be:

```
PRINT IN,N$
```

Each time that the program reaches this location the contents of IN and N\$ are displayed. The programmer then examines these values to see if they are what is expected. If not, this gives some indication of the source of the program error.

It is important to realize that this temporary trace output is displayed with other output. With a little experience, it is not too difficult to separate visually the normal output from the trace. If this becomes too much of a problem, some programmers temporarily remove the other print statements so the trace is easier to follow.

Often, the above trace methods are combined. For instance:

```
PRINT "ORDER QUANTITY",AC,Q
```

displays both a message and the contents of the variables AC (Account) and Q (Quantity) when this part of the program is reached.

**Sales Commission Program** To demonstrate this technique of debugging, we will write a sales commission program. In this program a salesperson is paid a commission based on a percentage of the dollar amount of sales. The percents are as follows:

Sales Amount	Commission Percentage
less than \$1000	5%
\$1000 to \$1999.99	7%
\$2000 to \$2999.99	10%
\$3000 or more	12%

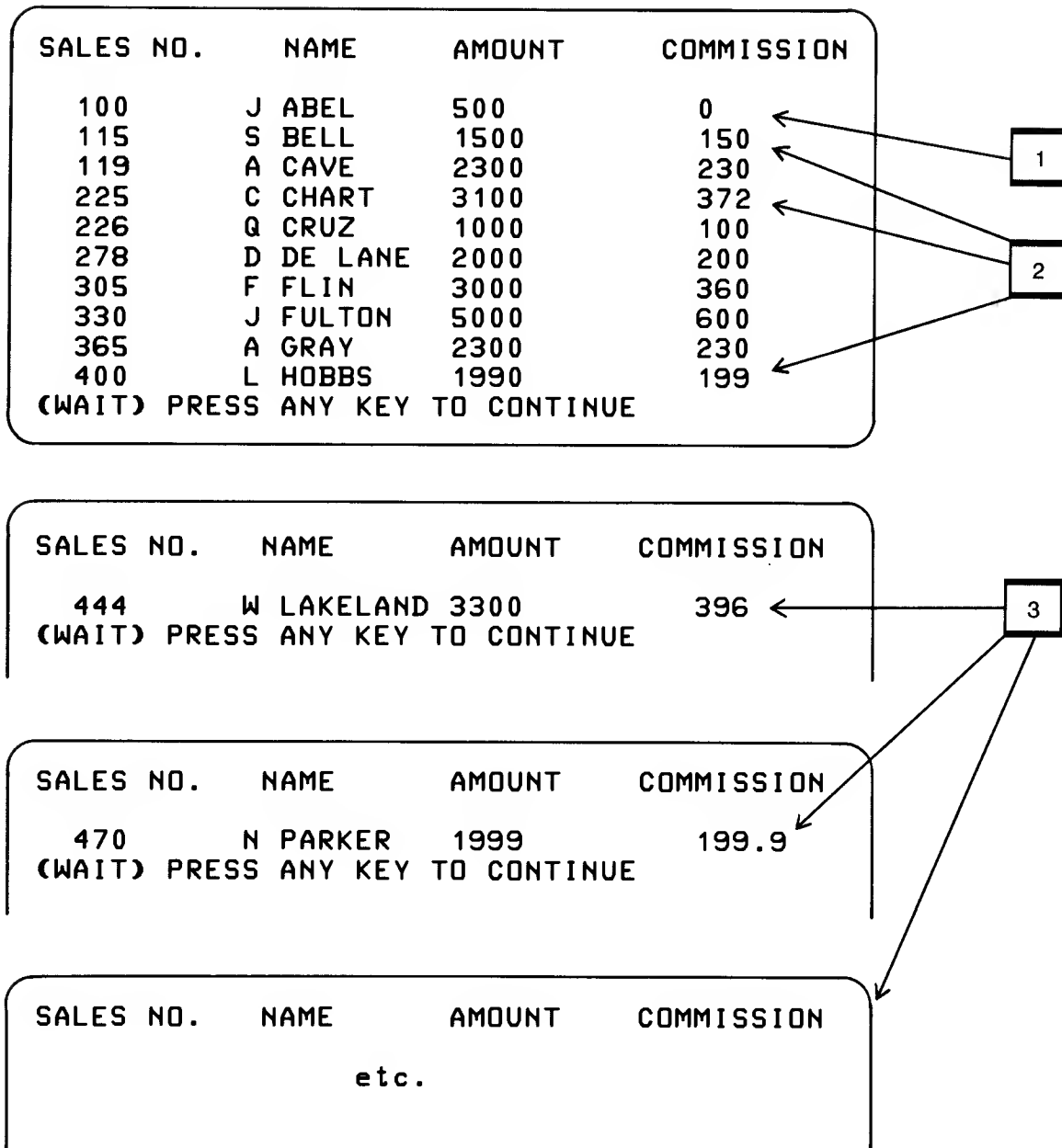
Figure 11.1 shows how the program and data might appear before the initial syntax errors have been corrected. Figure 11.2 shows the output that was printed on the first test run.

```

10 REM SALES COMMISSION WITH ERRORS
20 PRINT "[clr]"
30 PRINT "SALES NO.    NAME    AMOUNT    COMMISSION"
40 PRINT
50 READ S,N$,A
60 IF S=999 THEN 210
70 IF A<1000 THEN 130
80 IF A<2000 THEN 140
90 IF A<3000 THEN 160
100 C=A*.12
110 GOTO 170
120 C=A*.05
130 GOTO 170
140 C=A*.07
160 C=A*.10
170 PRINT TAB(3);S;TAB(12);N$;TAB(24);A;TAB(35);C
180 L=L+1
190 IF L<10 THEN 50
200 PRINT "(WAIT) PRESS ANY KEY TO CONTINUE"
210 GET W$:IF W$="" THEN 210
220 GOTO 20
300 DATA 100,"J ABEL",500
310 DATA 115,"S BELL",1500
320 DATA 119,"A CAVE",2300
330 DATA 225,"C CHART",3100
340 DATA 226,"Q CRUZ",1000
350 DATA 278,"D DE LANE",2000
360 DATA 305,"F FLIN",3000
370 DATA 330,"J FULTON",5000
380 DATA 365,"A GRAY",2300
390 DATA 400,"L HOBBS",1990
400 DATA 444,"W LAKELAND",3300
410 DATA 470,"N PARKER",1999
420 DATA 500,"R MOORE",4000
430 DATA 554,"F PRATT",3900
440 DATA 999,"LAST",0

```

**Figure 11.1** Sales commission program with logic errors



**Figure 11.2** Sales commission program output

Upon examination of the output, several errors are noted. The first, labelled (1), shows that Abel has no commission calculated. This could be due to a forgotten calculation or an error relating to the first data record.

The second error (2) shows Cruz with a \$100 commission when it should be \$70. This error also occurs for the Bell and Hobbs data, which are computed at 10% when the rate should be 7%.

Lastly, (3) indicates that the screen is cleared and a heading displayed for each line after Hobbs. The intent was to display ten lines and then wait for the user to review the lines visually. When the user presses a key to continue, the next ten lines should display. Instead only one line is shown.

An experienced programmer might be able to find these errors without a trace; however, we will use a trace to discover the source of the errors.

Since it seems likely the Abel commission is due to a calculation error a trace will be placed after the calculation as follows:

```
120 C=A*.05
125 PRINT "LT 1000 COMMISSION";C
```

Problem 2 suggests a difficulty with salespersons in the less-than-2000 category so another trace is placed after the 7% calculation.

```
140 C=A*.07
145 PRINT "LT 2000 COMMISSION";C
```

The last error is the problem with headings. Since L is used to count the number of lines, it should be printed each time we print and count a line. This is done with:

```
180 L=L+1
185 PRINT "LINE COUNT =",L
```

All of these trace statements are included for the next run of the program. The listing is shown in figure 11.3. To save space the data statements are deleted from this printout. Figure 11.4 shows the output from this test run that we will now consider.

```
10 REM SALES COMMISSION WITH TRACES
20 PRINT "[clr]"
30 PRINT "SALES NO.    NAME        AMOUNT    COMMISSION"
40 PRINT
50 READ S,N$,A
60 IF S=999 THEN 210
70 IF A<1000 THEN 130
80 IF A<2000 THEN 140
90 IF A<3000 THEN 160
100 C=A*.12
110 GOTO 170
120 C=A*.05
125 PRINT "LT 1000 COMMISSION";C ←
130 GOTO 170
140 C=A*.07
145 PRINT "LT 2000 COMMISSION";C ←
160 C=A*.10
170 PRINT TAB(3);S;TAB(12);N$;TAB(24);A;TAB(35);C
180 L=L+1
185 PRINT "LINE COUNT =",L ←
190 IF L<10 THEN 50
200 PRINT "(WAIT) PRESS ANY KEY TO CONTINUE"
210 GET W$:IF W$="" THEN 210
220 GOTO 20
```




Figure 11.3 Sales commission program with trace statements

When we examine the results of our trace on the screen (figure 11.4), we are led to the necessary corrections for the program. The first error was the commission of zero for Abel. If we look at the trace output for Abel it doesn't exist. This is a major clue for finding the error, which strongly suggests the program never reached statement 125, the trace statement. By the process of logical deduction we conclude that statement 120 was also not done.

Now the question we should ask ourselves is, What code is responsible for bringing us to statement 120? It is the statement that determines Abel has sales of less than \$1000. This is statement 70. If you examine this statement you will see that it goes to statement 130, not 120. This is the error. It is corrected as follows:

70 IF A<1000 THEN 120

SALES NO.	NAME	AMOUNT	COMMISSION	
100	J ABEL	500	0	← data line
LINE COUNT = 1 ←				
LT 2000 COMMISSION 105 ←				
115	S BELL	1500	150	
LINE COUNT = 2				
119	A CAVE	2300	230	
LINE COUNT = 3				
225	C CHART	3100	372	
LINE COUNT = 4				
LT 2000 COMMISSION 70				
226	Q CRUZ	1000	100	
LINE COUNT = 5				
278	D DE LANE	2000	200	
LINE COUNT = 6				
305	F FLIN	3000	360	
LINE COUNT = 7				
330	J FULTON	5000	600	
LINE COUNT = 8				
365	A GRAY	2300	230	
LINE COUNT = 9				
LT 2000 COMMISSION 139.3				
400	L HOBBS	1990	199	
LINE COUNT = 10				
(WAIT) PRESS ANY KEY TO CONTINUE				

SALES NO.	NAME	AMOUNT	COMMISSION
444	W LAKELAND	3300	396
LINE COUNT = 11			
(WAIT) PRESS ANY KEY TO CONTINUE			

SALES NO.	NAME	AMOUNT	COMMISSION
LT 2000 COMMISSION 139.93			
470	N PARKER	1999	199.9
LINE COUNT = 12			

Figure 11.4 Program and trace output

The second error does produce a trace output. It shows that the calculation for Cruz's commission was 70 but that 100 was displayed. Thus the calculation was right but the output was wrong. If we look at this part of the program, we see the 7% calculated in 140. The next statement is the trace followed by a calculation for 100%. Apparently we forgot the GOTO at 150.

150 GOTO 170

The last problem was the new screen display after ten lines. The trace shows a line count for each line of output beginning at 1 for the first line but proceeding to 10, then 11, and so on. Since we are allowing only ten lines per screen we obviously forgot to reset the line counter to zero.

Now we remove the trace statements and include the corrections discussed above. At this stage a final test run is done as shown in figures 11.5 and 11.6. If there is still some concern at this stage of testing about whether all of the errors have been found or if they have been properly corrected, the trace statements could be left in the program for one more run.

```
10 REM SALES COMMISSION PROGRAM
20 PRINT "[clr]"
30 PRINT "SALES NO.    NAME        AMOUNT    COMMISSION"
40 PRINT
50 READ S,N$,A
60 IF S=999 THEN 210
70 IF A<1000 THEN 120
80 IF A<2000 THEN 140
90 IF A<3000 THEN 160
100 C=A*.12
110 GOTO 170
120 C=A*.05
130 GOTO 170
140 C=A*.07
150 GOTO 170
160 C=A*.10
170 PRINT TAB(3);S;TAB(12);N$;TAB(24);A;TAB(35);C
180 L=L+1
190 IF L<10 THEN 50
200 PRINT "(WAIT) PRESS ANY KEY TO CONTINUE"
210 GET W$:IF W$="" THEN 210
215 L=0
220 GOTO 20
300 DATA 100,"J ABEL",500
310 DATA 115,"S BELL",1500
320 DATA 119,"A CAVE",2300
330 DATA 225,"C CHART",3100
340 DATA 226,"Q CRUZ",1000
350 DATA 278,"D DE LANE",2000
360 DATA 305,"F FLIN",3000
370 DATA 330,"J FULTON",5000
380 DATA 365,"A GRAY",2300
390 DATA 400,"L HOBBS",1990
400 DATA 444,"W LAKELAND",3300
410 DATA 470,"N PARKER",1999
420 DATA 500,"R MOORE",4000
430 DATA 554,"F PRATT",3900
440 DATA 999,"LAST",0
```

Figure 11.5 Sales commission program—debugged

SALES NO.	NAME	AMOUNT	COMMISSION
100	J ABEL	500	25
115	S BELL	1500	105
119	A CAVE	2300	230
225	C CHART	3100	372
226	Q CRUZ	1000	70
278	D DE LANE	2000	200
305	F FLIN	3000	360
330	J FULTON	5000	600
365	A GRAY	2300	230
400	L HOBBS	1990	139.3
(WAIT) PRESS ANY KEY TO CONTINUE			

SALES NO.	NAME	AMOUNT	COMMISSION
444	W LAKELAND	3300	396
470	N PARKER	1999	139.93
500	R MOORE	4000	480
554	F PRATT	3900	468

**Figure 11.6** Sales commission output—debugged

## **REVIEW QUESTIONS—CHAPTER 11**

1. What is meant by desk checking and when would it be used?
2. Describe what is meant by syntax errors. What is the usual reason for a syntax error?
3. See how many different kinds of syntax error messages you can create. What is the correction in each case?
4. Give eight considerations for preparing test data.
5. Why is volume of test data not an appropriate concern?
6. Explain how immediate mode operations may be used for debugging a program.
7. What are the two kinds of program tracing? When is it best to use each one?
8. What is a trace statement? How would one be used for debugging a program?
9. How do you know when a program is completely debugged? Can you ever be absolutely certain?

---

# Appendix A

---

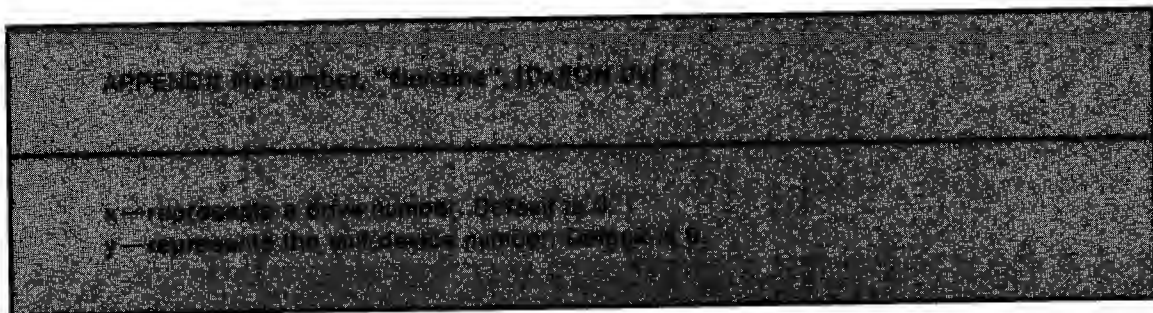
## BASIC

---

## Operating Commands

---

**T**his appendix lists some additional commands available in BASIC 4.0 on the PET and CBM. Most of these commands are used for disk systems but a few such as LIST, NEW, and RUN are available to all systems. The commands LOAD, SAVE, and VERIFY are used on tape systems.

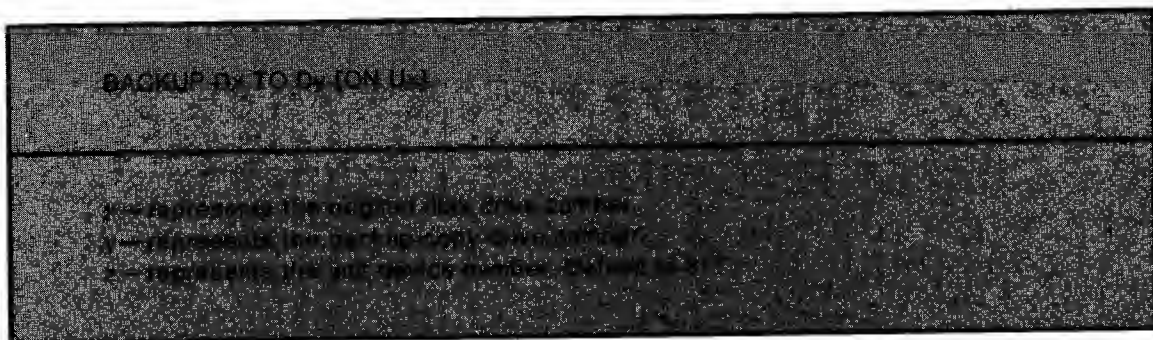


The APPEND command is used to write additional data to the end of a sequential disk file. APPEND is used like a DOPEN, but instead of writing over an existing file the PRINT# will add the new records to the end of the file.

Example:

```
100 APPEND#3,"CHECKBOOK",D1
```

Opens file CHECKBOOK as file number 3 on drive 1.



BACKUP creates a copy of an existing diskette on a second disk. This command is normally used in immediate mode.

Example:

```
BACKUP D0 TO D1
```

Copies the contents of the disk in drive 0 onto the disk in drive 1.



CATALOG [Dx] [ON Uy]

This is an optional command that is equivalent to asking for a DIRECTORY. See DIRECTORY for a detailed explanation.

CLR

The CLR command sets all numeric variables in a program to zero, all string variables to null (empty string), frees all array space, and resets memory and stack space.

Example:

10 CLR

If used at the beginning of the program all variables are cleared prior to program execution.

COLLECT [Dx] [ON Uy]

x—drive number. Default is last drive accessed.  
y—represents the unit device number. Default is 8.

COLLECT should be used occasionally to repair unallocated space on a disk. This space occurs due to improperly closed disk files. Normally COLLECT is used in immediate mode.

Example:

COLLECT D0

CONCAT [Dx,] "file1" TO [Dy,] "file2" [ON Uz]

x—disk drive for file 1.  
y—disk drive for file 2.  
z—represents the unit device number. Default is 8.

This command concatenates two sequential disk files. File 2 is replaced by the concatenated version of both files. File 1 is unchanged.

Example:

CONCAT D1,"MONTH" to D1,"ANNUAL"

The file named MONTH will be concatenated to the file named ANNUAL. ANNUAL now contains all of its original data plus the data contained on file MONTH. MONTH itself is unaffected by the operation.



The COPY command copies the contents of a file from one drive to another. The existing copy of the file is unchanged by the operation.

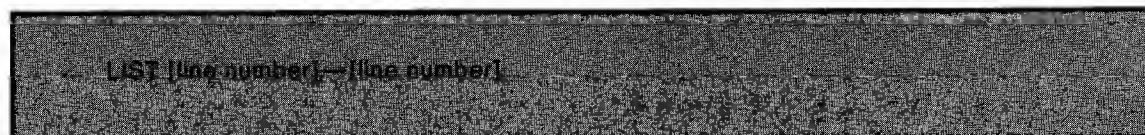
Example:

```
COPY "ACCOUNT.1",D0 TO "ACCOUNTS",D1
```

The file named ACCOUNT.1 on the floppy disk in drive 0 is copied to the disk in drive 1 and given the name ACCOUNTS.



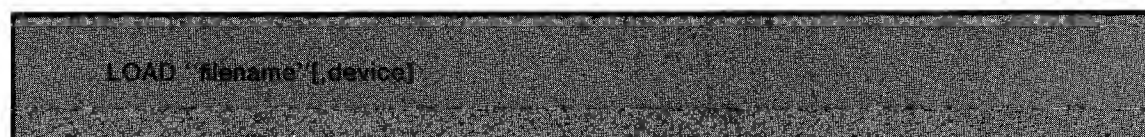
The DIRECTORY command gives a listing of the files on a diskette.



This command lists all or part of a BASIC program. When no line numbers are specified the entire program is listed. If a single line number is included only that line of the program is displayed. When two line numbers are included all statements in that range are listed on the screen.

Example:

```
LIST          list the entire program
LIST 150      displays line number 150
LIST 100-200  lists all statements from 100 to 200
```

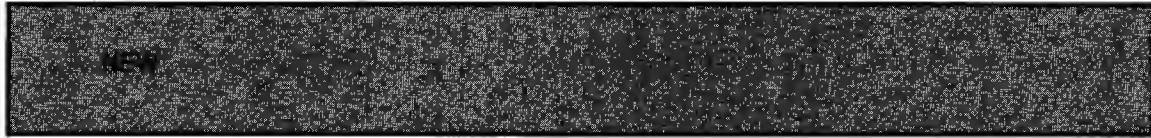


Normally the LOAD is used to load a program from tape, tape being the default when no device number is specified. Prior to BASIC 4.0 a LOAD using device number 8 was used for loading disk programs. It is recommended that DLOAD be used for disk.

Example:

**LOAD "CHASE"**

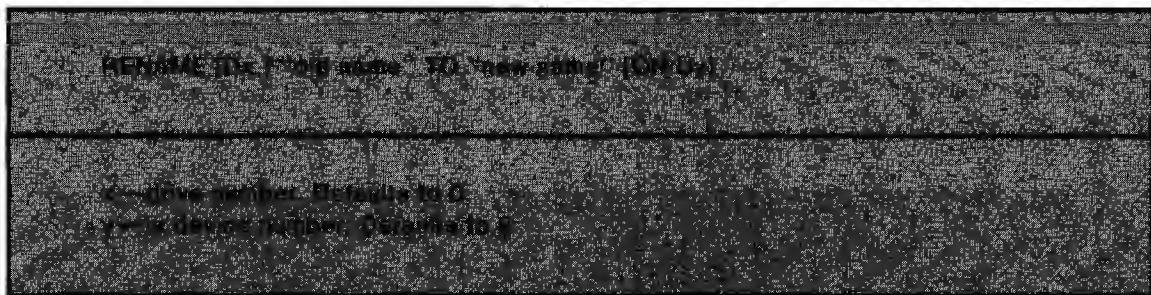
Loads the program CHASE from tape into memory.



NEW erases the current program from memory. This command should be used when you begin writing a new program.

Example:

**NEW**

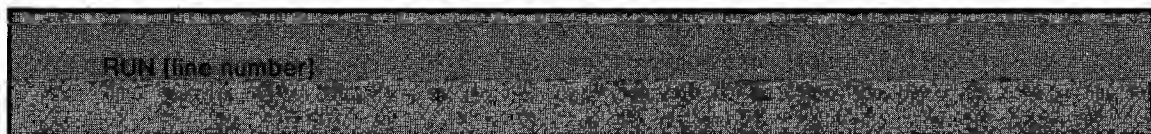


Normally used in immediate mode to change the name of a disk file. The old name of the file is replaced by the new name.

Example:

**RENAME "LEAPS" TO "BOUNDS"**

Changes the current filename LEAPS to BOUNDS.



Used to execute the program that is currently in memory. Including the line number begins execution at that line in the program.

Example:

**RUN**  
**RUN 100**

```
SAVE "filename" [device number]
```

Normally used to save a BASIC program on tape. By including a device number, programs may be saved on disk but DSAVE is recommended for use with disk.

Example:

```
SAVE "PHONE"
```

Saves the current BASIC program "PHONE" on tape.

```
SCRATCH "filename" [device number]
```

This command scratches or erases a file from disk. To ensure that you intend to scratch the file you must reply to the prompt "ARE YOU SURE?" with a YES response.

Example:

```
SCRATCH "EXPRESS",D1  
ARE YOU SURE? YES
```

Scratches file EXPRESS from the diskette in drive 1.

```
VERIFY "filename" [device number]
```

Used to verify the correctness of the contents of a program on tape. Normally VERIFY is used after the SAVE command to ensure the program was recorded accurately. After a program has been saved, the tape is rewound and the VERIFY command is entered.

Example:

```
VERIFY "PHONE"
```

---

## ***Appendix B***

---

### ***Reserved Words***

---

**C**ertain words in BASIC are intended to be used only for the purpose for which they were intended. Two-character words can present particular difficulties if they are used as variable names in a program.

ABS	GET	OPEN	SPC
AND	GET#	OR	SQR
ASC	GOSUB	PEEK	ST
ATN	GOTO	POKE	STEP
CHR\$	IF	POS	STOP
CLOSE	INPUT	PRINT	STR\$
CLR	INT	PRINT#	SYS
CMD	LEFT\$	READ	TAB
CONT	LEN	READ#	TAN
COS	LET	REM	THEN
DATA	LIST	RESTORE	TI
DEF	LOAD	RETURN	TI\$
DIM	LOG	RIGHT\$	TO
DS	MID\$	RND	USER
END	NEW	RUN	VAL
EXP	NEXT	SAVE	VERIFY
FN	NOT	SGN	WAIT
FOR	ON	SIN	
FRE			

# Appendix C

## Abbreviations

**E**arly in the book we used the question mark (?) to represent the command PRINT. The BASIC interpreter translated this abbreviation into the equivalent PRINT statement. Most reserved words in BASIC can be represented by a two- or three-letter abbreviation. When the abbreviation is two letters usually it will be the normal first letter of the reserved word followed by the second letter shifted. For example, the word LIST can be entered by typing L[shift] I. The shifted character will display as a graphic on the PET and an uppercase letter on the CBM.

In the following chart lowercase letters are typed normally while uppercase represents a shifted character.

WORD	ABBREV	WORD	ABBREV	WORD	ABBREV	WORD	ABBREV
ABS	aB	GET	gE	OPEN	oP	SPC	sP
AND	aN	GET#	get#	OR	or	SQR	sQ
ASC	aS	GOSUB	goS	PEEK	pE	ST	st
ATN	aT	GOTO	gO	POKE	pO	STEP	stE
CHR\$	cH	IF	if	POS	pos	STOP	sT
CLOSE	clO	INPUT	input	PRINT	?	STR\$	str\$
CLR	cL	INT	int	PRINT#	pR	SYS	sY
CMD	cM	LEFT\$	leF	READ	rE	TAB	tA
CONT	cO	LEN	len	READ#	read#	TAN	tan
COS	cos	LET	lE	REM	rem	THEN	tH
DATA	dA	LIST	lI	RESTORE	reS	TI	ti
DEF	dE	LOAD	lO	RETURN	reT	TI\$	ti\$
DIM	dI	LOG	log	RIGHT\$	rI	TO	to
END	eN	MID\$	mI	RND	rN	USER	uS
EXP	eX	NEW	new	RUN	rU	VAL	vA
FN	fn	NEXT	nE	SAVE	sA	VERIFY	vE
FOR	fO	NOT	nO	SGN	sG	WAIT	wA
FRE	fR	ON	on	SIN	sI		

---

# **Appendix D**

---

## **DOS**

---

### **Error Messages**

---

Type	Error Number	Error Message	Track	Sector
Status	00	OK	00	00
	01	FILES SCRATCHED	# Files	00
Read Errors	20	READ ERROR (Block header not found)	T	S
	21	READ ERROR (No sync character)	T	S
	22	READ ERROR (Data block not present)	T	S
	23	READ ERROR (Checksum error in data block)	T	S
	24	READ ERROR (Byte decoding error)	T	S
	27	READ ERROR (Checksum error in header)	T	S
Write Errors	25	WRITE ERROR (Write-verify error)	T	S
	26	WRITE PROTECT ON	T	S
	28	WRITE ERROR (Long data block)	T	S
	29	DISK ID MISMATCH	T	S
Syntax Errors	30	SYNTAX ERROR (General syntax)	00	00
	31	SYNTAX ERROR (Invalid command)	00	00
	32	SYNTAX ERROR (Long line)	00	00
	33	SYNTAX ERROR (Invalid filename)	00	00
	34	SYNTAX ERROR (No file given)	00	00
	39	SYNTAX ERROR (Invalid DOS command)	00	00
	50	SYNTAX ERROR (Record not present)	00	00
	51	SYNTAX ERROR (Overflow in record)	T	S
File Errors	52	SYNTAX ERROR (File too large)	T	S
	60	WRITE FILE OPEN	00	00
	61	FILE NOT OPEN	00	00
	62	FILE NOT FOUND	00	00
	63	FILE EXISTS	00	00
	64	FILE TYPE MISMATCH	00	00
	65	NO BLOCK	T	S
	66	ILLEGAL TRACK AND SECTOR	T	S
System Errors	67	ILLEGAL SYSTEM TRACK AND SECTOR	T	S
	70	NO CHANNEL	00	00
	71	DIR ERROR	00	00
	72	DISK FULL	00	00
	73	DOS MISMATCH	00	00
	74	DRIVE NOT READY	00	00

X X POKE 59468,12

X POKE 59468,14

## Appendix E

### PET ASCII

### and PEEK/POKE Codes

**T**he PET and CBM uses internal codes to represent all characters available to the programmer. One system of coding used is ASCII (American Standard Code for Information Interchange) although Commodore uses a variation of this standard. The ASCII codes are given by the ASC function and used by the CHR\$ function.

These codes are useful for producing screen graphics, in which case the PEEK or POKE value is used from the right column of the chart.

ASCII	PEEK	ASCII	PEEK	ASCII	PEEK	ASCII	PEEK	ASCII	PEEK	ASCII	PEEK
163	# 99	165	% 101	184	8 120	183	7 119	177	1 113	214	V 86
197	E 69	212	T 84	185	9 121	175	/ 111	178	2 114	219	[ 91
196	D 68	199	G 71	181	5 117	180	4 116	179	3 115	206	N 78
195	C 67	194	B 66	182	6 118	170	* 106	171	+ 107	205	M 77
192	@ 64	221	J 93								
198	F 70	200	H 72							166	& 102
210	R 82	217	Y 89	161	/ 97	169	) 105	209	Q 81	220	\ 92
164	\$ 100	167	' 103	162	// 98	223	← 95	215	W 87	168	( 104
193	A 65	213	U 85	207	O 79	188	< 124	176	Ø 112		
218	Z 90	202	J 74	204	L 76	190	> 126	174	• 110		
211	S 83	201	I 73	208	P 80	172	, 108	173	- 109		
216	X 88	203	K 75	186	: 122	187	; 123	189	= 125		
						191	? 127				



































# Appendix F






































## ASCII

### and CHR\$ Codes

**T**his appendix shows you what characters will appear if you PRINT CHR\$ (X), for all possible values of X. It will also show the values obtained by typing PRINT ASC ("x") where x is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper-/lowercase, and printing character-based commands (like switch to upper-/lowercase) that could not be enclosed in quotes.

PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$	PRINTS	CHR\$
	0		16	SPACE	32	Ø	48
	1	CRSR	17	!	33	1	49
	2	RVS ON	18	"	34	2	50
	3	CLR HOME	19	#	35	3	51
	4	INST DEL	20	\$	36	4	52
WHT	5		21	%	37	5	53
	6		22	&	38	6	54
	7		23	.	39	7	55
	8		24	(	40	8	56
	9		25	)	41	9	57
	10		26	*	42	:	58
	11		27	+	43	;	59
	12	RED	28	,	44	<	60
RETURN	13	CRSR	29	—	45	=	61
SWITCH TO LOWER CASE	14	GRN	30	.	46	>	62
	15	BLU	31	/	47	?	63

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
@	64	U	85	 *	106	 ?	127
A	65	V	86	 +	107		128
B	66	W	87	 ,	108		129
C	67	X	88	 ~	109		130
D	68	Y	89	 .	110		131
E	69	Z	90	 /	111		132
F	70	[	91	 Ø	112	f1 <i>120</i>	133
G	71	<i>£ \</i>	92	 1	113	f3	134
H	72	]	93	 2	114	f5 <i>Bel</i>	135
I	73	↑	94	 3	115	f7 <i>bl</i>	136
J	74	←	95	 4	116	f2 <i>bl</i>	137
K	75	 <i>black</i>	96	 5	117	f4 <i>bl</i>	138
L	76	 !	97	 6	118	f6 <i>bl</i>	139
M	77	 "	98	 7	119	f8 <i>bl</i>	140
N	78	 #	99	 8	120	SHIFT	141
O	79	 \$	100	 9	121	RETURN	142
P	80	 %	101	 :	122	SWITCH TO UPPER CASE	143
Q	81	 &	102	 ;	123	BLK	144
R	82	 '	103	 <	124	CRSR	145
S	83	 (	104	 =	125	RVS OFF	146
T	84	 )	105	 >	126	CLR HOME	147

PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS	PRINTS	CHRS
	148		159		170		181
	149		160		171		182
	150		161		172		183
	151		162		173		184
	152		163		174		185
	153		164		175		186
	154		165		176		187
	155		166		177		188
	156		167		178		189
	157		168		179		190
	158		169		180		191

# Index

- abbreviation, 187
- ABS, 87
- add, 23
- AND, 44
- animation, 123–28
- APPEND#, 179
- appending, 159, 179
- arithmetic functions, 87
- arithmetic statements, 23
- arrays, 39, 69, 152
- ASC, 90
- ASCII, 82, 90, 136, 138, 191, 193
- ATN, 87
  
- BACKUP, 179
- backup, 13, 179
- bar charts, 60
- BASIC 4.0, 3, 15, 144
- bit, 18
- budget program, 137, 141, 146
- byte, 18
  
- CAI chess, 99
- calculator mode, 15, 19
- carriage return, 136
- cassette. *see* tape
- CATALOG, 12, 180
- CBM, 1, 5, 8, 12, 81, 131
- checkbook program, 149–55
- chessboard graphic, 123
- CHR\$, 91, 135, 138, 147, 193
- CLEAR, 17
- clear screen, 26, 32, 63
- clock. *see* TI; TI\$
- CLOSE, 134
- CLR, 180
- COLLECT, 180
- comma, 25
- command language, 115
- computer assisted instruction, 62, 93, 99
- CONCAT, 180
- concatenation, 94
- COPY, 181
- COS, 87
- cursor controls, 8, 16, 63, 111, 123
  
- DATA, 59
- DCLOSE, 145
- debugging, 165, 168, 173
- decimal positions, controlling, 86, 98
- defaults, 111
- DEF FN, 85
- degrees to radians, 90
- DELETE, 17
- DIM, 39
- DIRECTORY, 12, 181
- disk, 2, 11–13
- disk errors, 147
- disk files, 143
- divide, 24
- DLOAD, 12
- DOPEN, 144, 156
- DOS 2.0, 4, 143, 189
- driver routines, 98
- DS, 148
- DS\$, 148
- DSAVE, 12
  
- egg timer, 125
- end of file, 61, 134, 146
- English code, 29, 33, 36, 49, 52, 55, 150
- EXP, 88
- exponentiation, 24
  
- Fahrenheit-Celsius program, 30, 32
- filename, 134, 144, 149
- files, 133, 143, 149, 155
- file updating, 139, 158
- floating point, 20
- flowcharts, 66, 67, 73, 140
- form filling, 114
- FOR-NEXT, 44, 50
- functions, 87, 90
  
- GET, 77, 113
- GOSUB, 46, 78
- GOTO, 28, 79
- Graphics, 7, 117
- Graphs, 95
  
- hardware, 1
- hierarchy, 24
- HOME, 8, 17
  
- IF, 42
- immediate mode, 15, 19, 168
- index file, 160
- INPUT, 27, 110
- input, 1, 27, 54
- INPUT#, 136, 145, 157
- INSERT, 16
- instring search, 102
- integer names, 21
- integers, 18
- interest program, 85
- INT function, 48, 88
  
- K, 3
- keyboard, 7
- keyboard graphic, 120
  
- LEFT\$, 91
- LEN, 92
- LIST, 181
- LOAD, 10, 181
- load error, 10
- loan payments program, 35
- LOG, 88
- logical operators, 43
- lowercase, 81
- lunar lander graphic, 122
  
- memory, 2, 17
- menus, 112–14
- metric conversion program, 66
- MID\$, 92
- multiple fields, 138, 147
- multiple statements, 22, 43
- multiply, 23
- music player, 131
  
- NEW, 17, 182
- number guessing game, 49, 51
- number pad, 8
- numbers, 18

- OFF/RVS, 8
- ON GOSUB, 78
- ON GOTO, 79
- OPEN, 134
- OR, 44
- output, 1
- parentheses, 24
- payroll program, 70
- PEEK, 82, 191
- PET, 1, 5
- plotting graphs, 95
- POKE, 80, 91, 119, 121, 125, 127, 129, 191
- PRINT, 15, 25, 63, 110, 117
- PRINT#, 135, 145
- print zones, 25
- program, 1, 10
- program generalization, 99
- prompting, 110
- pseudo code, 29, 33, 36, 49, 52, 55, 150
- radians to degrees, 90
- RAM, 2
- random numbers, 48, 83, 89
- random responses, 102
- reaction timer, 120
- READ, 59
- real numbers, 18
- RECORD#, 157
- relative access files, 155

- REM, 29
- RENAME, 182
- replacing a file, 149
- replacing a program, 12
- reserved words, 185
- RESTORE, 62
- RETURN, 46
- return key, 7, 27
- RIGHT\$, 94
- RND function, 48, 88
- rocket animation, 123, 124
- rolling die, 127
- ROM, 2, 7
- rounding, 18
- RUN, 10, 31, 182
- RUN/STOP, 8, 17
- sales commission program, 173
- SAVE, 11, 183
- scientific notation, 20
- SCRATCH, 183
- screen addresses, 81
- screen characters, 91
- scrolling, 12, 151
- semicolon, 26, 28, 30, 45
- sequential files, 133, 144
- SGN, 89
- simple calculation program, 29
- SIN, 89, 95
- sound, 129
- SPC, 65

- SQR, 90
- ST, 137
- STOP, 8, 17, 47
- STR\$, 94
- string functions, 90
- string names, 22
- strings, 21
- software, 1
- subtract, 23
- SuperPET, 1
- syntax errors, 166

- TAB, 65
- TAN, 90
- tape, 2, 9–11, 133, 136
- test data, 164
- time delays, 50, 129
- TI function, 48, 83, 126
- TI\$ function, 84
- tracing, 168, 172, 176
- trip costs program, 53
- truncating, 19

- updating. *see* file updating
- uppercase, 81

- VAL, 94
- variable names, 21
- variables, 21
- VERIFY, 11, 183

- weighted average program, 32, 52, 61